# Chapitre 1

## Introduction

### 1.1 Revue générale

Avec les changements rapides de notre société en une société d'informations, de plus en plus de gens se fient sur la technologie de l'information pour vivre et pour travailler. Ceci a entraîné, dans le passé récent, le développement des réseaux informatiques et donc des protocoles de communication. Brièvement, un protocole de communication définit un ensemble de règles avec lesquelles des ordinateurs hétérogènes (construits par différents vendeurs) peuvent échanger de l'information afin de fournir des services de communication aux usagers.

Avec le besoin croissant pour des applications ainsi que pour des réseaux plus fiables, il y a une demande croissante pour des protocoles de communication qui peuvent être utilisés, d'une manière fiable, pour l'échange de données entre les applications et les éléments du réseau. A part la spécification et la normalisation de ces protocoles, le test de conformité des implantations des protocoles par rapport aux spécifications constitue un aspect très important de la garantie de la qualité. Ainsi, la vérification formelle et les approches de test ont été développées en se basant sur les techniques de description formelle *TDF*s. Plusieurs modèles ont été étudiés pour tester la conformité des protocoles de communication. Le modèle le plus utilisé est le modèle de machine à états finis ou *FSM (Finite State Machine)*. Ce dernier a été initialement introduit pour la synthèse des circuits digitaux synchrones. Il a été ensuite utilisé pour la spécification des protocoles de communication [Boch 87, Ural 91, Sidh 89, Toll 90] etc.

Les *FSM*s peuvent être étendues de plusieurs façons afin de prendre en compte les différents aspects à modéliser. Les aspects les plus demandés sont les données dues à l'extension d'applications dont l'aspect prépondérant est la donnée comme par exemple les applications multimedia. Le temps constitue un autre aspect qui modélise les applications à caractère temps réel. L' extension du modèle *FSM* qui est utilisée de plus en plus pour spécifier ce type d'application (e.g., les logiciels de communication) est le modèle de machines à états finis étendus ou *EFSM (Extended Finite State Machine)*. D'ailleurs, certaines *TDF*s sont basées sur ce modèle telles que *SDL* [Belina 89], *LOTOS* [Bolo 87] et *Estelle* [Budk 86]. Cependant, ce modèle peut ne pas convenir à la description des protocoles de communication du fait qu'ils sont en général formés de plusieurs processus communicants. Le modèle approprié dans ce cas est le modèle de *EFSM* communicante ou *CEFSM* (Communicating *EFSM*).

## 1.2 Le test des protocoles de communication

Afin d'être sûr de construire des réseaux fiables, nous devons nous assurer que les systèmes se comportent et échangent leurs informations d'une manière correcte. Comme ces systèmes peuvent être issus de vendeurs différents, ceci n'est possible que si des règles générales sont établies pour les procédures de communication. Ainsi, des organisations internationales telles que *ISO* (International Standardisation Organization), *ITU* (International Telecommunications Union, connu sous *CCITT*) et l'*ETSI* (European Telecommunications Standards Institute) travaillent pour la normalisation des protocoles de communication.

Le test de conformité des implantations de protocoles par rapport aux spécifications joue un rôle très important, du point de vue du développeur comme du point de vue de la normalisation. Le test peut seulement montrer la présence d'erreurs et pas leur absence, il ne peut donc pas être utilisé pour prouver l'exactitude, à moins d'être utilisé pour prouver l'exactitude par rapport à un certain modèle de fautes, i.e, en considérant certaines classes de fautes. Cependant, le test peut être, et est utilisé pour augmenter la confiance dans le fonctionnement exact d'une implantation d'un protocole. Du point de vue du développeur, ceci est important du fait que ça donne quelques indications sur la qualité du produit.

Normalement, des procédures de test sont utilisées pour décider si des implantations peuvent être livrées. Les organisations de normalisation (L'*ISO*, l'*ITU* et l'*ETSI*) quant à elles, livrent des standards de suites de test pour la conformité des protocoles, i.e., spécifient un nombre de tests. Ces suites de test doivent être utilisées pour la certification des implantations de protocoles. La certification est utile dans un environnement de marché où l'on peut choisir entre des implantations de protocoles de différents vendeurs.

Le développement de suites de test est un processus complexe, lent et sujet à des erreurs. Pour ces raisons, des activités de recherche ont été entreprises pour développer des méthodes de génération automatique de tests. La plupart de celles-ci sont basées sur des techniques telles que décrites dans [Koha 78]. Ces techniques sont principalement conçues pour le test de matériel et sont basées sur le modèle de machine à états finis (*FSM*). Pour les techniques de spécification telles que *SDL* et *LOTOS*, des méthodologies de test formelles ont été développées. Cependant, ces méthodes ne peuvent pas être complètement automatisées ou ne peuvent être appliquées dans la pratique.

Les méthodes de génération de test existantes mettent l'emphase sur le comportement dynamique des protocoles et négligent de traiter la partie *données*. Cependant, pour de plus en plus de protocoles, la complexité de la partie données est de beaucoup supérieure à celle du comportement dynamique. Le test de la partie données pour les implantations de ces protocoles doit donc être prépondérant. Pour cette raison, les méthodes de génération de cas de test se concentrant sur les données sont nécessaires.

## 1.3 La normalisation des protocoles de communication

Les méthodes de génération de cas de test pour les protocoles de communication sont développées et décrites en utilisant un cadre de travail spécial pour le test. Ce cadre de travail est normalisé dans *ISO*. Il se base sur le modèle de référence de base pour l'interconnexion des systèmes ouverts (ou modèle *OSI* pour Open Systems Interconnection). Ce dernier offre une classification des fonctions du réseau; il est aussi utilisé dans le développement et la normalisation des protocoles de communication. Nous devons noter que d'autres modèles existent. Cependant, les concepts de base ainsi que la vue générale, par rapport aux protocoles de communication, sont partagés par ces modèles.

Le modèle *OSI* est décrit dans la section 1.3.1. Un aspect typique de ce modèle est la distinction qui est faite entre *services* et *protocoles*. Une discussion de cet aspect est présentée dans la section 1.3.2. La section 1.3.3 conclut la section 1.3 en décrivant comment et quels aspects sont décrits dans les standards des protocoles de communication.

### 1.3.1 Le modèle de référence de base OSI

Afin de faciliter le développement et la normalisation des protocoles de communication, *ISO* a développé le modèle de référence de base *OSI* [Day 83]. Ce modèle est normalisé [ISO-7498 84] et offre une classification des fonctions du réseau. Cette classification est réalisée en définissant une pile de sept couches dans lesquelles différentes fonctions du réseau résident. Dans ce modèle, les fonctions qui résident dans la couche N dépendent des fonctions qui résident dans une couche, généralement la couche N-1, au dessous de la couche N.

Une représentation graphique du modèle de référence de base de l'*OSI*, montrant les sept couches et leur organisation, est montrée dans la figure 1.1. Les différentes couches de l'*OSI* sont caractérisées comme suit:

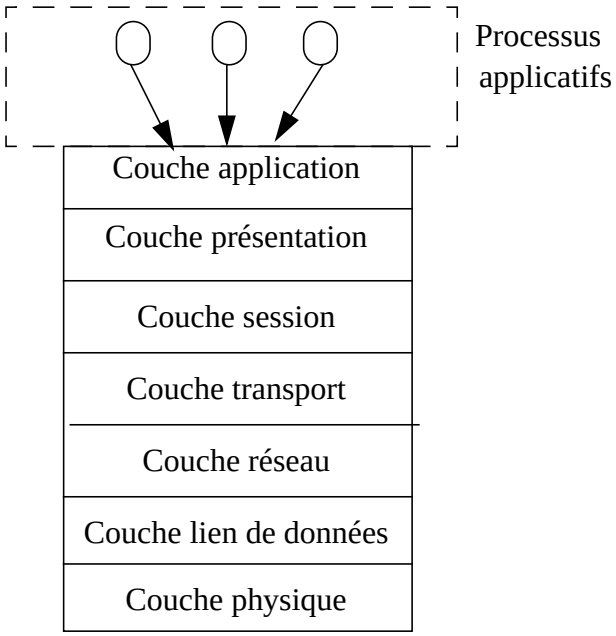



Figure 1.1 Le modèle de référence de base OSI

**Description des couches**

- La **couche physique** offre la transmission transparente de blocs de bits à travers des connexions physiques. Elle couvre aussi les connexions physiques, e.g., câbles, modems et autres interfaces et incorpore les fonctions d'activation et de désactivation mécaniques électromagnétiques des connexions physiques.

- La **couche lien de données** offre les moyens pour la transmission des bits à travers un lien logique unique entre les entités du réseau. Elle incorpore aussi des fonctions pour détecter et corriger les erreurs dans les blocs de bits.

- La **couche réseau** s'occupe d'acheminer les paquets de données à un noeud destination à travers un réseau de noeuds (fonction de routage). Elle incorpore aussi les fonctions pour la fragmentation, le ré-assemblage ainsi que pour le contrôle de flux et de congestion.

- La **couche transport** agit comme une interface entre les couches supérieures et inférieures. Elle assure un transfert de données fiable et transparent et dégage ses processus utilisateurs de tout souci concernant la façon avec laquelle un transfert de données efficace est réalisé. Elle est capable de transférer des données d'un processus sur un système à un processus sur un autre système (vision bout en bout entre les processus pairs).

- La **couche session** offre les moyens nécessaires à l'organisation et à la synchronisation du dialogue de l'échange de données. Elle garde trace des besoins spéciaux de connexion, i.e., initialisation des paramètres de ses processus utilisateurs. Elle offre donc une connexion virtuelle à ses processus utilisateurs.

- La **couche présentation** offre les moyens pour la présentation de l'information que ses processus utilisateurs utilisent dans leur communication. Elle rend possible la communication entre des systèmes utilisant des représentations internes différentes de l'information. Elle incorpore également les fonction de cryptage (sécurité des données) et de compression.

- La **couche application** offre des services de support aux processus applicatifs réels pour accéder à l'environnement *OSI*. Ces derniers sont utilisés par des processus application distribués. Elle incorpore plusieurs protocoles qui sont réalisés par les éléments de service application.

Il est à noter que le modèle *OSI* est un modèle abstrait. Ceci veut dire par exemple qu'il y a seulement une couche *transport*. Tous les processus qui implantent les fonctions et les procédures de la couche transport résident donc dans la même couche, même s'ils s'exécutent sur des systèmes différents.

Les standards des protocoles de communication, émis par les organisations internationales telles que *ISO, ETSI* et *ITU,* décrivent seulement les règles pour les communications distantes. Ceci est dû au fait que tous les aspects locaux de communication peuvent être implantés sur un système, rendant la normalisation inutile voire indésirable. Les constructeurs des systèmes sont libres de choisir leurs propres

solutions quant à la manière de gérer les communications locales. Pour des besoins de performance, un constructeur peut décider de combiner des processus issus de couches différentes en un seul processus. En résumé, les standards des protocoles de communication établissent des règles pour la communication entre des entités qui résident dans une seule couche (la même), sachant que ces entités peuvent être implantées dans des entités différentes du réseau. Les standards des protocoles de communication ne mettent aucune contrainte sur les aspects locaux de communication. Les règles pour les communications entre des entités qui résident dans la même couche, et donc qui peuvent être implantées sur différentes entités du réseau, définissent un *protocole de communication*. Les aspects de communication entre les entités qui résident dans des couches adjacentes, i.e., aspects locaux de communication, définissent *le service de communication*. Par conséquent, dans les descriptions des standards, une distinction nette est faite entre les *services* et les *protocoles* du fait que le protocole est normalisé alors que le service ne l'est pas.

### 1.3.2 Les services et les protocoles

Dans le contexte du modèle *OSI*, un service est un ensemble de primitives de communication, par exemple, des appels de fonctions ou de procédures, qui implantent la fonctionnalité d'une couche spécifique. Ces primitives sont utilisées par tout processus désirant utiliser les fonctions et procédures de la couche.

Le terme "Protocole" (pour une procédure de communication de données) fût utilisé pour la première fois par R. A. Scantlebury et K. A. Baretlett au Laboratoire National Physique en Angleterre en 1967 comme cité dans [Holz 91]. Un protocole de communication est un ensemble de formats et de règles régissant la communication d'entités par échange de messages. La définition complète d'un protocole ressemble beaucoup à la définition d'un langage:

- il définit un format précis pour les messages valides: une syntaxe,
- il définit les règles de procédure pour l'échange de données: une grammaire,

- il définit un vocabulaire pour les messages (appelés *UDPs*: Unités de données du protocole) valides qui peuvent être échangés, avec leur signification: une sémantique.

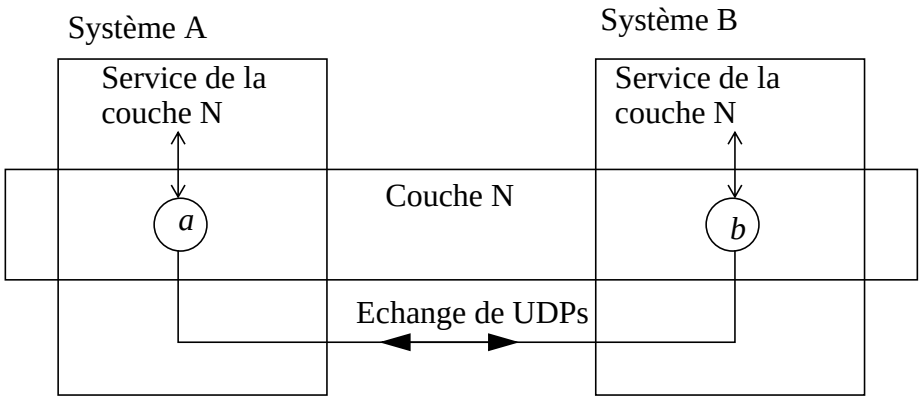



Figure 1.2 Vue architecturale du modèle *OSI*

Un modèle architectural typique pour voir la réalisation de la fonctionnalité d'une couche N du modèle *OSI* est présenté dans la figure 1.2. Cette dernière montre deux systèmes A et B, qui exécutent des processus résidant à la couche N, i.e., le processus *a* et le processus *b.* Le processus *a* offre à tout processus du système A le service de la couche N, alors que le processus *b* fait la même chose dans le système B (on les nomme des processus homologues ou processus pairs). Les processus *a* et *b* échangent de l'information à travers les *UDP.* L'échange de ces *UDP* est réalisé par le service de la couche N-1.

### 1.3.3 Les standards de communication

Les standards des protocoles de communication émis par *ISO* et *ITU* peuvent être vus comme des spécifications d'entités qui implantent (une partie de) la fonctionnalité d'une couche spécifique du modèle *OSI*. Pour les couches inférieures, la fonctionnalité est généralement implantée par une seule entité. Dans ces standards, les entités qui résident dans les couches sont généralement appelées *machines de protocoles*. Les aspects

importants de ces dernières, qui sont aussi décrits dans les standards de protocoles, sont le comportement dynamique, qui peut inclure les *aspects de temps* et les *temporisateurs*, et les *données* qui comprennent la syntaxe des unités de données et les relations entre les unités de données.

Pour certains de ces aspects, les standards incorporent des descriptions formelles. La description du comportement dynamique est normalement réalisée par des tables d'états, qui ressemblent aux machines à états finis, des *MSCs* (Message Sequence Charts) ou bien par des processus *SDL*; *MSC* étant un langage de trace pour les systèmes de communication; il décrit des parties de l'exécution des systèmes. La syntaxe des unités de données peut être décrite par *ASN 1* (Abstract Syntax Notation number 1) [ISO-8824].

## 1.4 Résumé

Le besoin croissant des réseaux de communication avec des applications distribuées élaborées pousse au développement de protocoles de communication de plus en plus complexes. Pour permettre l'échange d'information dans des réseaux de différents vendeurs, ces protocoles sont normalisés par des organisations internationales (*ITU*, *ISO*, etc). Les standards de communication sont décrits en se basant sur le modèle de référence de base *OSI*. Afin d'améliorer la qualité, et réduire le temps pour développer des suites de tests, les intérêts de recherche se sont penchés sur les techniques de génération automatique de cas de test. Actuellement, les techniques existantes sont basées sur le modèle de *FSM*. Par conséquent, les tests des aspects de données ne peuvent être générés. Aussi, les techniques se basant sur les *EFSM*s permettent de résoudre ce problème mais deviennent inefficaces pour les systèmes modélisés par plusieurs processus communicants.

## 1.5 Objectifs de cette thèse

Notre but est de développer une méthode pratique pour générer automatiquement des cas de test pour les protocoles de communication afin de décider si une implantation du protocole est conforme à sa spécification. Comme mentionné auparavant, il existe des méthodes de génération de cas de test, cependant celles-ci ont plusieurs limitations. La plupart de ces méthodes extraient la *FSM* correspondante au système et génèrent des cas

de test pour tester le flux de contrôle seulement. D'autres méthodes permettent de tester le flux de données des systèmes, mais celles-ci ne génèrent des cas de test que pour des systèmes modélisés par une seule *EFSM* ou ignorent les boucles. Dans le cas où le système est modélisé par plusieurs *EFSM*s qui communiquent entre elles et avec l'environnement, ces méthodes génèrent le graphe d'accessibilité, qui représente le comportement global du système, et génèrent des cas de test pour tester les flux de contrôle et de données de ce dernier. Cette méthode n'est malheureusement pas toujours applicable à cause du problème de l'explosion combinatoire (ou explosion d'états). Aussi, certaines méthodes de génération de cas de test qui s'intéressent au test du flux de données ne peuvent pas tester certains genres de spécifications, notamment, celles qui contiennent des boucles dont le nombre d'itérations est inconnu au départ ou dépend de certains paramètres en entrée.

Notre but est de développer une méthode de test qui considère les deux aspects *contrôle* et *données*. Cette méthode teste les protocoles de communication ou tout système pouvant être modélisé par une ou plusieurs *EFSMs* tout en évitant l'explosion combinatoire. Pour plus de détail, notre méthode permet de:

- Tester les systèmes modélisés par une *EFSM:* flux de contrôle et de données
- Tester les systèmes modélisés par plusieurs *EFSM* communicantes via une technique de propagation
- Tester les systèmes dont la spécification contient des boucles non bornées
- Calculer la couverture des cas de test
- Générer les séquences d'entrée/sortie (E/S) de taille raisonnable afin de pouvoir faire le test de conformité
- Tester les systèmes spécifiés par *SDL*.

## 1.6 Plan de la thèse

Cette thèse est divisée en deux parties; une première partie, constituée des chapitres 2, 3 et 4, relatant l'état de l'art et une deuxième partie, contenant les chapitres restants, présentant notre contribution.

Nous considérons que les logiciels de communication sont des cas particuliers de logiciels généraux. Les différentes phases du cycle de développement des logiciels sont résumées dans le deuxième chapitre. Cependant, afin de pouvoir tester un logiciel de communication (ou tout système), ce dernier doit d'abord être spécifié. Pour ce faire, plusieurs modèles existent. Le plus connu est le modèle de *FSM* qui est plus approprié pour le test du flux de contrôle des systèmes. Dans ce chapitre, nous parlons également des modèles utilisés pour la spécification des systèmes, notamment le modèle de *FSM* et ses extensions. Aussi, le modèle des *réseaux de* Petri ainsi que la logique formelle qui constituent d'autres formalismes pour la spécification des systèmes, sont présentés. Finalement, certains langages de spécification tels que le langage *SDL*, *Estelle*, *StateCharts* et *LOTOS* son décrits.

Dans le troisième chapitre, nous parlons du test de logiciel, qui est une des plus importantes activités dans le cycle de développement de logiciels. Nous définissons les différentes stratégies de test, les types de test, les architectures de test et la couverture de test.

Dans le quatrième chapitre, les méthodes existantes de génération de cas de test sont décrites. Ainsi, nous présentons les méthodes dédiées au test du flux de contrôle et du flux de données. Et puisque nous nous intéressons à générer des cas de test à partir de spécifications *SDL*, nous présentons un panorama d'outils de génération de cas de test à partir de spécifications écrites dans ce langage. Il est à noter que la plupart de ces outils sont semi-automatiques et nécessitent que les buts de test soient spécifiés par l'utilisateur.

Ainsi, après avoir fait le tour des méthodes existantes de génération de cas de test, nous entamons la deuxième partie de cette thèse en présentant dans le cinquième chapitre notre méthode de génération de cas de test pour les systèmes modélisés par une seule

*EFSM*. Cette méthode est une méthode unifiée pour le test du flux de contrôle et du flux de données.

Généralement, les spécifications des protocoles de communication et des systèmes en général sont constituées de plus d'une *EFSM*. La méthode présentée dans le cinquième chapitre n'est donc plus suffisante. Pour cette raison, nous avons développé une méthodologie de génération de cas de test pour les systèmes modélisés par plusieurs *CEFSMs*, que nous présentons dans le sixième chapitre. Cette méthodologie teste un système en générant des cas de test pour chacune de ses *CEFSMs* dans le contexte, i.e., en considérant sa communication avec les autres *CEFSM*s. Au lieu de générer le graphe d'accessibilité pour le système global, ce dernier est testé d'une manière incrémentale afin d'éviter le problème de l'explosion combinatoire. Cette méthodologie est pratique et permet donc de tester les systèmes réels.

Le septième chapitre, quant à lui, présente l'architecture de l'outil *CEFTG* (Communicating Extended Finite state machine Test Generator) ainsi que ses principales composantes.

Le huitième et dernier chapitre résume tout ce qui a été présenté dans cette thèse et présente les extensions possibles de ce travail.

# Chapitre 2

## Le génie logiciel et la phase de spécification

### 2.1 Introduction

Le génie logiciel existe depuis trente ans environ. Il a été défini pour répondre à un problème qui devenait de plus en plus évident. D'une part, le logiciel n'était pas fiable, d'autre part, il était très difficile de réaliser dans des délais prévus des logiciels satisfaisant leurs cahiers de charges.

Malgré le fait que plusieurs auteurs ont développé des définitions personnelles du génie logiciel, une définition proposée par Fritz Bauer dans [Naur 69] sert toujours de base aux discussions:

"*Le génie logiciel est l'établissement et l'utilisation de principes d'ingénierie solides afin d'obtenir d'une manière économique un logiciel qui est fiable et qui marche efficacement sur des machines réelles*".

Le génie logiciel est donc l'art de spécifier, de concevoir, de réaliser, et de faire évoluer, avec des moyens et dans des délais raisonnables, des programmes, des documentations et des procédures de qualité en vue d'utiliser un ordinateur pour résoudre un problème.

Le génie logiciel est une technologie en couches (voir figure 2.1). La base du génie logiciel est la couche *processus*. La couche processus du génie logiciel est la colle qui tient la technologie des couches ensemble et qui permet un développement rationnel des

logiciels.

La couche des méthodes offre la technique “des comment” pour la construction de logiciels. Les méthodes englobent une large gamme de tâches qui inclut: l’analyse des besoins, la conception, la construction de programmes, le test et la maintenance. Les méthodes de génie logiciel reposent sur un ensemble de principes de base qui gouvernent chaque région de la technologie et incluent les activités de modélisation et d’autres techniques descriptives.

Les outils du génie logiciel offrent un support automatique ou semi-automatique pour le processus et les méthodes. Quand les outils sont intégrés, un système pour le support du développement du logiciel, appelé *CASE* (Computer-Aided Software Engineering) est établi. *CASE* comprend du logiciel, du matériel et une base de données de génie logiciel (contenant des données importantes sur l’analyse, la conception, le test, etc) pour créer un environnement de génie logiciel qui est analogue au *CAD/CAE* (Computer-Aided Design/ Engineering) pour le matériel.

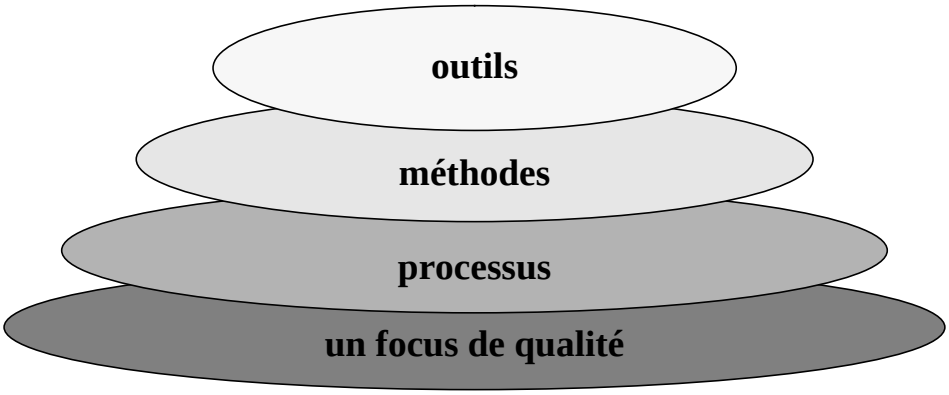



Figure 2.1 Les couches du génie logiciel

Le génie logiciel incorpore une stratégie de développement qui comprend le processus, les méthodes et les outils décrits plus haut. Cette stratégie est souvent appelée *modèle de processus* ou *paradigme de génie logiciel*. Un modèle de processus est choisi en fonction de la nature du projet et de l’application, des outils et méthodes à utiliser et des

contrôles et délivrables.

Le modèle du cycle de vie (ou de processus) d'un logiciel est un modèle de phases ou activités qui commence quand le logiciel est conçu et se termine quand le produit n'est plus disponible pour l'utilisation. Le cycle de vie d'un logiciel comprend typiquement une phase des besoins, une phase de conception, une phase d'implantation (codage et tests unitaires), une phase d'intégration et de test, une phase d'installation et de vérification et une phase d'opération et de maintenance. Selon le cycle de vie choisi, ces phases ou activités peuvent survenir une seule fois dans l'ordre indiqué ou plusieurs fois dans un autre ordre.

Plusieurs modèles de cycle de vie d'un logiciel existent (le modèle en cascade [Royc 70], prototypage rapide [Goum 81], prototypage évolutif [Gidd 84], réutilisation de logiciel [Jone 84], développement incremental [Hirs 85], etc.).

Actuellement, pour les applications nouvelles, le mode d'organisation le plus employé et normalisé par l'*AFNOR* (Association Francaise de Normalisation) est une technique par anticipation appelée Modèle en V schématisée par la figure ci-après.

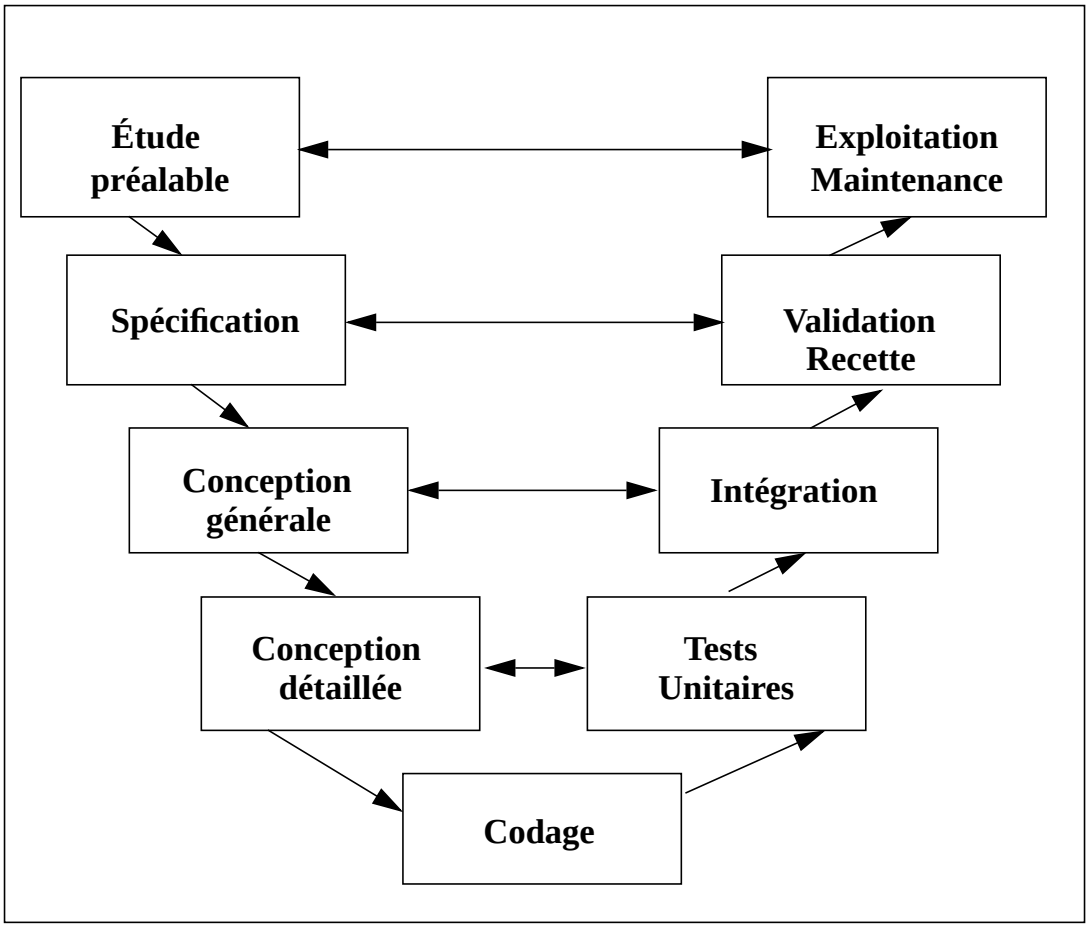



Figure 2.2 Le modèle de développement en V

Un élément fondamental du schéma en V est constitué par les flèches horizontales qui traduisent un mode de fonctionnement standard qui se retrouve à tous les étages du développement: la phase qui se trouve dans la branche de gauche doit élaborer les procédures de test et validation qui seront exécutées dans la phase située au même niveau dans la branche de droite.

Cette procédure évite de tomber dans le piège classique qui consiste à valider le résultat par rapport à ce qui aurait dû être fait. Elle permet d'éviter en outre de nombreux problèmes contractuels ou commerciaux entre le producteur et le client.

Le développement des logiciels passe par des phases qui amènent à la production d'un système vérifiant certaines caractéristiques et répondant aux besoins préalablement requis. Ces phases font partie de tous les cycles de développement de systèmes indépendamment de la nature, du domaine, de la taille et de la complexité du système à développer.

Après avoir présenté le cycle de vie d'un logiciel, nous allons parler dans ce qui suit de la phase de spécification, et plus particulièrement, des méthodes de spécification des protocoles de communication qui sont des cas particuliers de logiciels.

## 2.2 Les méthodes de spécification des protocoles

Les méthodologies de conception ne sont pas très bien définies aux haut niveaux d'abstraction. Il serait donc préférable de dévouer plus d'efforts à la spécification de la fonctionnalité du système au tout début du processus avant qu'aucune décision de conception ne soit prise.

L'utilisation de la spécification écrite dans le langage naturel semble être comprise, mais celle-ci entraîne souvent des spécifications informelles et longues qui contiennent des ambiguïtés et pour lesquelles il est difficile de vérifier la complétude et l'exactitude. Dans ce contexte, plusieurs techniques de description formelles ont été proposées, incluant les machines à états finis, les réseaux de *Petri*, les langages de programmation de haut niveau, la logique temporelle, etc.

La spécification des protocoles est typiquement divisée en deux parties: une partie *contrôle* modélisée le plus souvent par une machine à états finis et une partie *données* modélisée par des segments de programmes.

Dans ce qui suit, nous présentons le modèle de machine à états finis (*FSM*), les extensions de ce modèle ainsi que d'autres méthodes de spécification de systèmes.

## 2.3 Machines à états finis

Avec les technologies avancées des ordinateurs, les systèmes deviennent de plus en plus gros, afin d'assumer des tâches de plus en plus difficiles. Par conséquent, le test

constitue une partie indispensable pour les phases de conception et d'implantation, et cependant, il s'avère que c'est une tâche très ardue pour les systèmes complexes. Ceci a motivé l'étude des machines à états finis pour assurer un fonctionnement exact des systèmes et pour découvrir certains aspects de leurs comportements.

A un niveau d'abstraction bas, un système est souvent mieux compris comme une machine à états. Les critères de conception peuvent aussi être facilement exprimés en termes d'états ou transitions du système désirables ou indésirables. Dans un sens, l'état du système symbolise les hypothèses que chaque processus du système fait sur les autres. Il définit quelles sont les actions qu'un processus peut prendre, quels événements attendre et comment répondre à ces événements.

Une machine à états finis (*FSM*) est un modèle abstrait pour décrire un système dont le comportement passé peut affecter le comportement futur d'une multitude de façons. Ce modèle a été très étudié dans la théorie des automates des années 50 jusqu'aux années 70, pour le développement de circuits séquentiels synchrones [Gill 62, Koha 78]. Il a été aussi utilisé dans d'autres domaines tels que l'analyse lexicale, les réseaux itératifs, etc. A partir des années 80, ce modèle a été exploré dans le but de l'appliquer au développement des protocoles de communication, en général, et pour le test de conformité de ces protocoles en particulier. Plus récemment, le modèle de FSM est utilisé pour la conception et le test des systèmes orientés objets [Hoff 93].

Les automates à états finis sont bien adaptés à la spécification de logiciels et particulièrement les protocoles de communication. Les *FSM*s modélisent les modules du système sous forme d'un graphe étiqueté, où les noeuds représentent l'état du système (valeur des variables, contenu des files d'attentes, etc) et les arcs représentent les transitions ou les événements internes ou externes qui, lorsqu'ils sont traités, peuvent changer l'état du système. Chaque module est représenté par une *FSM* et le comportement global du système peut être obtenu sous certaines contraintes par composition des *FSM*s. Cette méthode de spécification est une méthode mathématique, elle est non ambiguë et est facilement automatisable.

Dans le modèle de machines à états finis traditionnel, l'environnement de la machine consiste en deux ensembles de signaux finis et disjoints: les signaux en entrée et les signaux en sortie. Un signal peut avoir un ensemble fini de valeurs possibles. La condition qui doit être satisfaite pour que la transition soit exécutable est alors exprimée sous forme de condition sur les valeurs de chaque signal en entrée, et l'effet de la transition peut changer les valeurs des signaux en sortie.

Il y a deux types de *FSM*: les machines Mealy [Meal 55] et les machines Moore [Moor 64]. La théorie est similaire pour les deux types. Nous considérons dans ce qui suit des machines Mealy; ces dernières modélisent les systèmes à états finis d'une façon plus appropriée. Une machine Mealy possède un nombre fini d'états et produit des sorties après la réception de données d'entrée.

Formellement, une *FSM* est un 6-tuple F = < I, O, S, $s_i$, $\delta$, $\lambda$ > où

- I est un ensemble fini de symboles d'entrée,
- O est un ensemble fini de symboles de sortie,
- S est un ensemble fini d'états,
- $s_I \in S$ est l'état initial,
- $\delta: S \times I \rightarrow S$ est la fonction de transfert (aussi appelée fonction ProchainEtat),
- $\lambda: S \times I \rightarrow O$ est la fonction de sortie.

Quand la machine est à l'état courant s de S et reçoit une donnée d'entrée a de I, elle passe au prochain état spécifié par $\delta(s, a)$ et produit la sortie donnée par $\lambda(s, a)$. Pour les machines Moore, la fonction de sortie dépend seulement des états et non pas des états et des données d'entrée.

La fonction de transfert et la fonction de sortie caractérisent ensemble le comportement de la *FSM*. On doit noter que si le domaine de la spécification *D= S x I*, alors la fonction de transfert $\delta$ et la fonction de sortie $\lambda$ sont définies pour toutes les

combinaisons état-donnée d'entrée et dans ce cas, la *FSM* est dite complètement spécifiée ou complètement définie.

Dans [Bisw 1986], les auteurs présentent une représentation utile des *FSMs* (table de transition d'états) pour construire des suites de tests selon le standard *OSI* pour le test de conformité. Un exemple de table de transition d'états est donné dans la table 1.

| **État courant** | **Entrée** | **Sortie** | **Prochain état** |
|---|---|---|---|
| q0 | - | 1 | q2 |
| q1 | - | 0 | q0 |
| q2 | 0 | 0 | q3 |
| q2 | 1 | 0 | q1 |
| q3 | 0 | 0 | q0 |
| q3 | 1 | 0 | q1 |

**TABLE 1. Table de transition d'états**

Le comportement de la machine présentée dans la table 1 est plus facile à comprendre quand cette dernière est représentée graphiquement sous forme de diagramme de transition d'états, comme montré dans la figure 2.3.

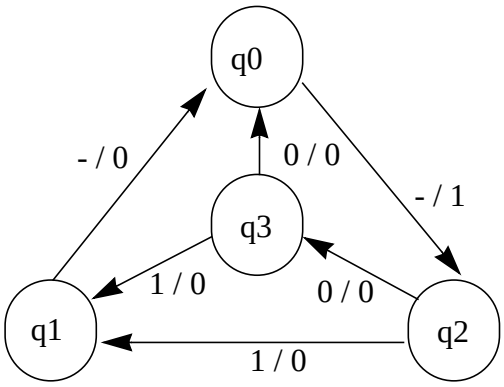



Figure 2.3 Diagramme de transition d'états

**Définition 4.1**: Une *FSM* M est initialement connectée si tous ses états sont accessibles à partir de l'état initial.

**Définition 4.2**: M est dite fortement connectée si, pour chaque paire d'états ($s_i$, $s_j$), il y a une sequence x de données d'entrée qui change l'état de la machine de l'état $s_i$ à $s_j$.

## 2.4 Machines à états finis non-déterministes

Une *FSM* non-déterministe (*NFSM*) est une machine Mealy qui peut être formellement définie comme suit:

Une *NFSM* A est un 6-tuple (S, I, O, h, $s_i$, D) où

- I est en ensemble fini de symboles d'entrée,
- O est en ensemble fini de symboles de sortie,
- S est un ensemble fini d'états,
- $s_i \in S$ est l'état initial,
- $D \subseteq SxI$ est le domaine de la spécification,
- h est une fonction de comportement qui définit les transitions possibles de la machine.

Pour chaque état courant $s_m$ et donnée d'entrée x, chaque paire ($s_{mn}$,y) dans le résultat de la fonction représente une transition possible amenant au prochain état $s_n$ et produisant la donnée de sortie y.

La machine A devient déterministe si $|h(s, x)| = 1$ pour tout $(s, x) \in L$. Comme mentionné auparavant, dans une machine déterministe, on utilise deux fonctions: la fonction ProchainEtat $\delta$, et la fonction de sortie $\lambda$, par contre, dans une *NFSM*, la fonction de comportement est utilisée. La *NFSM* est dite complètement spécifiée si $D \subset SxI$. Un exemple de machine non-déterministe est présenté dans la table 2.

| État courant | Entrée | Sortie | Prochain état |
|---|---|---|---|
| q1 | - | 0 | q0 |
| q1 | 0 | 0 | q3 |

**TABLE 2. Exemple de machine à états finis non-déterministe**

Nous avons présenté le modèle de *FSM*, mais ce dernier peut ne pas être adéquat pour représenter certains systèmes, notamment ceux qui contiennent plusieurs processus qui communiquent, par des messages, entre eux et avec l'environnement. Avant de voir un des modèles utilisés pour représenter ce type de systèmes, nous définissons dans la prochaine section les deux mécanismes de communication les plus populaires étant donné que chaque modèle se base sur un mécanisme différent.

## 2.5 Communication synchrone versus communication asynchrone

Dans un premier temps, nous définissons quelques critères des mécanismes de communication existants [Andr 83].

- Processus émetteur bloquant ou non-bloquant: on dit qu'un processus émetteur est bloquant quand il arrête d'envoyer des signaux si la zone tampon du receveur, espace mémoire utilisé pour recevoir le ou les messages, est pleine. Dans le cas contraire, on parle de processus émetteur non bloquant.

- Processus receveur bloquant ou non bloquant: on dit qu'un processus receveur est bloquant quand il arrête d'exécuter des transitions si sa file de signaux, espace mémoire utilisé pour récupérer les messages, est vide. Dans le cas contraire, on parle de processus receveur non bloquant.

- Type de la zone tampon: les caractéristiques des zones tampons sont intéressantes, i.e., si la zone tampon emmagasine les signaux selon un ordre *FIFO* (First in First Out ou premier arrivé premier servi). La zone tampon est appelée *FIFO* si l'ordre de consommation des signaux est identique à l'ordre de leur envoi. Si par contre, les signaux peuvent être consommés d'une manière arbitraire la zone tampon est appelée "multi-set buffer". Si une zone tampon ne peut emmagasiner qu'un seul signal, i.e, les autres signaux sont ignorés, elle est appelée "set buffer".

- Communication synchrone est une communication impliquant deux processus qui ne possèdent pas de zone tampon. Les processus émetteur et receveur sont tous les deux bloquants. Dans un modèle synchrone, à chaque instant, chaque processus exécute une seule étape. Ceci suggère l'idée d'une horloge globale qui donne un rythme aux comportements. Ce type de communication est utilisé dans le langage *LOTOS*.

- Communication asynchrone avec *FIFO* est une communication entre deux processus tels que le processus émetteur est non-bloquant et le processus receveur est bloquant. Ce type de communication est utilisé dans les langages *SDL* et *Estelle*.

Il existe d'autres types de communication asynchrone (avec "multi-set buffer" ou "set buffer") mais nous n'avons présenté que la communication asynchrone avec *FIFO* étant donné que c'est celle qui nous intéresse dans cette thèse.

## 2.6 Systèmes communicants

Une *FSM* communicante (*CFSM*) [Holz 91] peut être définie comme une machine abstraite qui accepte des données d'entrée, génère des données de sortie et change son état interne selon un plan pré-défini. Les machines *FSM* communiquent via des files *FIFO* bornées qui transforment les données de sortie d'une machine en données d'entrée d'une autre. Définissons tout d'abord formellement le concept de file.

Une file à messages est un triplet (S, N, C), où:

- S est un ensemble fini appelé vocabulaire de la file,

- N est un entier qui définit le nombre de places dans la file, et

- C est le contenu de la file, un ensemble ordonné d'éléments de S.

Les éléments de S et de C sont appelés messages. Chaque message porte un nom unique mais représente un objet abstrait. Soit M l'ensemble de toutes les files, un exposant $1 \le m \le |M|$ est utilisé pour identifier chaque file, et un indice $1 \le n \le N$ est utilisé pour identifier un espace dans la file. $C_m^n$ est le nième message dans la m$^{\text{ième}}$ file.

Un vocabulaire de système V peut être défini comme la réunion de tous les vocabulaires de toutes les files, plus un élément nul que l'on va représenter par $\varepsilon$. Avec l'ensemble de files M, numérotées de 1 à $|M|$, le vocabulaire du système V est défini comme $V = \bigcup_{m=1}^{|M|} S^m \cup \varepsilon$ .

Maintenant, définissons formellement une *CFSM*. Une *CFSM* est un tuple (Q, $q_0$, M, T) où:

- Q est un ensemble fini, non vide d'états,
- $q_0$ est un élément de Q, l'état initial,
- M est un ensemble de files, comme définies plus haut, et
- T est une relation de transitions.

La relation T prend deux arguments q et a où q est l'état courant et a est une action. Jusque là, trois types d'actions sont permises: l'entrée des données, la sortie des données et l'action nulle $\varepsilon$. L'occurrence des deux premiers types d'actions dépend de l'état des files. Si elles sont exécutées, elles changeront l'état d'une seule file.

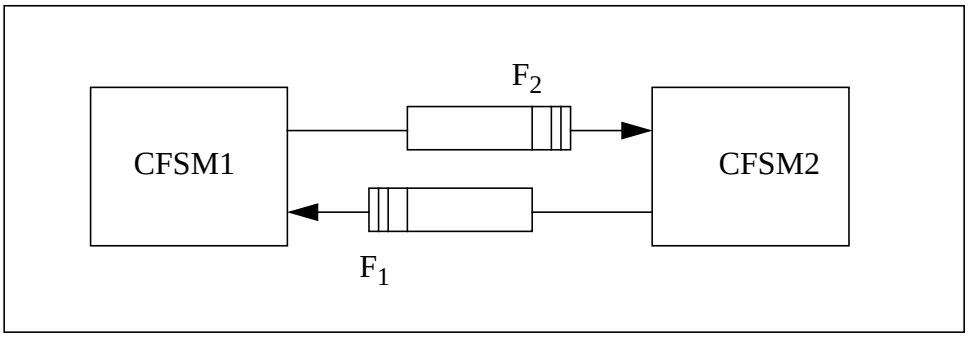



Figure 2.4 Exemple de système communicant formé de deux *CFSMs*

Dans la figure 2.4, $F_i$ représente la file d'attente de la *CFSM*$_i$.

La relation de transition T définit un ensemble de zéro ou plusieurs états successeurs possibles dans l'ensemble Q pour l'état courant q. Cet ensemble contiendra précisément un seul état, sauf si le non-déterminisme est modélisé. Si T(q, a) n'est pas définie explicitement, on suppose que $T(q, a)= \varnothing$.

$T(q,\varepsilon)$ spécifie les transitions spontanées. Une condition suffisante pour que ces transitions soient exécutables est que la machine soit à l'état q.

Définissons maintenant ce qu'est un système communicant.

**Définition 4.3**: Un système communicant $\Pi$ de n *CFSMs* est un 2*n tuplet ($C_1$, $C_2$,..., $C_n$, $F_1$, $F_2$,..., $F_n$) où

- Ci est une *CFSM*
- $F_i$ est une file d'attente *FIFO* pour $C_i$, i=1..n.

Supposons que le système $\Pi$ consiste en n *CFSM*s communicantes: $C_1$, $C_2$,...,$C_n$. Les *CFSM*s échangent des messages à travers des files d'attente *FIFO* de capacité finie. Nous supposons qu'une *FIFO* existe pour chaque *CFSM* et que tous les messages vers cette *CFSM* passent à travers cette *FIFO*. Nous supposons aussi dans ce cas qu'un message interne identifie son émetteur et son récepteur. Un message en entrée peut être interne s'il est envoyé par une autre *CFSM* ou externe s'il arrive de l'environnement.

### 2.6.1 Analyse d'accessibilité

**Définition 4.4**: Un état global de $\Pi$ est un 2*n-tuple $<s^{(1)}, s^{(2)},..., s^{(n)},m_1, m_2,...,m_n>$ où $m_j$, j=1...n sont des ensembles de messages contenus dans $F_1$, $F_2$,..., $F_n$ respectivement.

**Définition 4.5**: Une transition globale de $\Pi$ est une paire $t = (i,\alpha)$ où $\alpha \in A_i$ est un ensemble d'actions. Les actions peuvent être soit des messages entrants, sortants ou des événements internes. Une transition t est exécutable (déclenchable, ou tirable) dans s = $<s^{(1)}, s^{(2)},..., s^{(n)}, m_1, m_2,..., m_n>$ si et seulement si les deux conditions suivantes sont satis-

faites où $\alpha$ = (etat_initial, etat-final, entrée, prédicat, bloc-calcul):

- Une relation de transitions est définie $\delta_i$ $(s,\alpha)$
- $input = null$ et $predicat = Vrai$ ou $input = a$ et $m_i = aW$, où W est un ensemble de messages destinés à $C_i$, et $Predicat = Vrai$.

Après l'exécution de la transition t, le système passe à l'état s' = $<s'^{(1)}, s'^{(2)},..., s'^{(n)}, m'_1, m'_2,..., m'_n>$ et les messages contenus dans les canaux sont $m'_j$ où

- $s'(i) = \delta(s(i),\alpha)$ et $s'(j) = s(j) \quad \forall (j \neq i)$
- si $input = \varnothing$ et $output = \varnothing$, alors $m'_j = m_j$
- si $input = \varnothing$ et $output = b$ alors $m'_k = m_k b$ ($C_k$ est la machine qui reçoit b)
- si $input \neq \varnothing$ et $output = \varnothing$ alors $m_i = W$ et $m'_j = m_j \ \forall (j \neq i)$
- si $input \neq \varnothing$ et $output = b$ alors $m_i = W$ et $m'_k = m_k b$.

On dit alors que l'état $s_2$ est accessible à partir de l'état $s_1$ et on note: $s_1 \rightarrow_t s_2$. Soit $s_0 = <s_0^{(1)}, s_0^{(2)},..., s_0^{(k)}, \varepsilon, \varepsilon,..., \varepsilon>$ l'état initial du système communicant. On dit qu'un état g du système $\Pi$ est accessible s'il est accessible de l'état initial $s_0$.

Les états accessibles de $\Pi$ constituent les noeuds du *graphe d'accessibilité* de $\Pi$. Ce dernier est noté $R_\Pi$ et la procédure pour le construire est appelée *analyse d'accessibilité.*

Le modèle obtenu à partir d'un système communicant par analyse d'accessibilité est appelé un modèle global. Ce modèle est un graphe orienté G = (V, E) où V est un ensemble d'états globaux et E correspond à l'ensemble de transitions globales.

### 2.6.2 Propriétés de communication

La validation d'un protocole par l'analyse d'accessibilité consiste à développer le graphe d'accessibilité du système communicant correspondant, puis à examiner les états globaux et la structure du graphe obtenu à la lumière d'un certain nombre de propriétés.

Pour définir ces dernières, nous allons utiliser des systèmes communicants formés uniquement de deux *CFSM*s.

Interblocages

Un interblocage est un état $s = (s_1,s_2,m_{1,2},m_{2,1})$ où $m_{1,2} = m_{2,1} = \varepsilon$ et pour i = 1, 2 $\delta_i\ (s_i,\alpha)$ est indéfinie $\forall \alpha \in A$.

Réceptions non spécifiées

On dit qu'il existe une réception non spécifiée pour une *CFSM* $C_i$ dans un état global $s = (s_1, s_2, m_{1,2}, m_{2,1})$ si $m_{j,i} = \alpha m'_{j,i}$ et $\delta_i\ (s_i,\alpha)$ est indéfinie.

Réceptions non spécifiées bloquantes

On dit qu'il existe une réception non spécifiée bloquante pour une *CFSM* $C_i$ dans un état global $s = (s_1, s_2, m_{1,2}, m_{2,1})$ si $\delta_i\ (s_i,\alpha)$ est indéfinie pour tout $\alpha$ (message sortant de $C_i$ ou événement interne) et $m_{j,i} = \alpha\ m'_{j,i}$ avec $\alpha \in M_{j,i}$ et $\delta_i\ (s_i,\alpha)$ est indéfinie.

Boucles bloquantes

Dans un système communicant $\Pi$, un cycle bloquant est un ensemble d'états globaux accessibles $B = \{s_1, s_2,..., s_n,...\}$ tel que:

- $|B| > 1$
- $s_0 \notin B$ ($s_0$ état l'état initial)
- si $(s_1,s_2) \in B$ alors $s_1 \rightarrow s_2$
- si $s_1 \in B$ et $s_2 \notin B$ alors $\neg(s_1 \rightarrow s_2)$.

Dépassement de la capacité des canaux

Quand la capacité d'un canal (file d'attente entre deux *CFSM*s) est définie par l'entier n ( $n > 0$ ), on dit qu'il y a dépassement de capacité dans un état global

s=($s_1$,$s_2$,$m_{1,2}$,$m_{2,1}$) si $|Fi| = n$ et il existe un message $\alpha$ (message sortant de $C_i$) tel que $\delta_i$ ($s_i$,$\alpha$) est définie.

D'après tout ce qui précède, nous pouvons conclure que la construction d'un graphe d'accessibilité $R_\Pi$ du système $\Pi$ est simple:

- On tire toutes les transitions tirables dans l'état initial $s_0$,
- On tire ensuite toutes les transitions tirables dans les états globaux obtenus,
- On continue ainsi jusqu'à ce qu'il soit impossible de créer un nouvel état global.

Le premier problème de cet algorithme est sa terminaison. En effet, si la taille des canaux de communication n'est pas bornée, les messages peuvent s'accumuler indéfiniment dans les canaux rendant le graphe d'accessibilité de taille infinie. Cependant, dans la pratique, la taille des canaux est finie, ce qui résout le premier problème. En supposant que le graphe soit fini, le deuxième problème important relatif à l'analyse d'accessibilité est l'explosition combinatoire, ce qui rend la mise en oeuvre de cette méthode sur les systèmes informatiques quasiment impossible. Des méthodes d'analyse d'accessibilité réduite sont proposées pour remédier à ces problèmes.

### 2.6.3 Analyse d'accessibilité réduite

L'analyse d'accessibilité réduite se fixe comme objectif de développer un graphe plus petit que le graphe d'accessibilité correspondant, permettant de montrer au moins une occurrence de certaines propriétés de communication. Ainsi, si l'analyse d'accessibilité permet d'effectuer une validation complète d'un système communicant vis à vis d'un certain nombre de propriétés, l'analyse d'accessibilité réduite est à l'évidence plus efficace pour montrer l'absence de propriétés.

Nous présentons ci-dessous, brièvement, trois méthodes d'analyse d'accessibilité réduite.

Méthode de Rubin & West [Rubi 82]

Appelée aussi analyse d'accessibilité par paire d'actions, cette méthode construit le graphe $R_\Pi$ d'états globaux $s = < s^{(1)}, s^{(2)}, m_1, m_2 >$ d'un système $\Pi$ formé de deux machines communicantes telles que $\|m_1\| - \|m_2\| = 0$. Une transition ne porte plus sur une seule action mais une paire d'actions appartenant aux deux machines. On ne peut transiter d'un état à un autre que si chacune des deux machines peut exécuter une action.

Cette méthode ne prend pas en compte les événements internes, elle détecte tous les interblocages et est difficilement réalisable pour les systèmes ayant plus que deux machines communicantes.

Méthode de Gouda & Yu [Goud 84]

Appelée aussi analyse d'accessibilité par progression maximale, cette méthode consiste en une analyse en deux phases produisant chacune un graphe d'accessibilité correspondant. Pour un système $\Pi = (C_1, C_2, F_1, F_2)$ composé de deux machines communicantes, la priorité est donnée à $C_1$ dans la première phase. Ainsi, quand deux transitions $t_1$ de $C_1$ et $t_2$ de $C_2$ sont tirables dans un état global, la priorité est donnée à $t_1$, la machine $C_2$ aura la priorité dans la deuxième phase.

Cette méthode est utilisée pour la détection des dépassements de capacité des canaux; ceci est d'ailleurs le point fort de cette méthode. En effet, s'il y a dépassement de capacité d'un canal, ceci sera sûrement détecté dans l'une des deux phases. Cette méthode ne prend pas en compte les événements internes, détecte au moins un interblocage, une réception non spécifiée et une réception non spécifiée bloquante s'ils existent. Cette méthode est difficilement généralisable pour des systèmes ayant plus que deux machines communicantes.

Méthode de Lee [Lee 96]

[Lee 96] est une méthode pour le test de conformité des protocoles modélisés par des *CFSM*s. Un état d'une *CFSM* peut être un *état stable global* ou un *état*

*transitoire*. Dans le premier cas, l'état est observable et dans le second il ne l'est pas. Les états et les transitions sont classifiées selon ce qui suit:

- Classification des états: un état est testé s'il a été observé au moins une fois durant le test. Sinon, il est non-testé. Un état est donc testé s'il a fait partie d'un état stable global ayant été observé.

- Classification des transitions: une transition avec une donnée d'entrée externe est:

  - non-testée, si elle n'a jamais été exécutée

  - faiblement testée, si elle a été exécutée et si son état final n'a pas encore été observé

  - testée, si elle a été exécutée et si son état final est observé.

La procédure de génération de cas de test est adaptive. Lorsqu'on arrive à un état stable global, on observe tous ses états composants, on identifie toutes les transitions sortantes possibles, on choisit une donnée d'entrée (à partir de ces transitions) et on continue l'expérience. La clé ici consiste à choisir la donnée d'entrée. L'idée de base de cette approche est simple: quand on choisit la prochaine donnée d'entrée, on choisit toujours une transition externe qui n'a pas encore été testée (selon la classification des transitions). S'il y a plusieurs transitions externes non encore testées, une transition est choisie au hasard. Ceci peut être vu comme une *visite aléatoire guidée*: on prend une décision selon la classe des transitions comme guide; mais parmi les transitions d'une même classe, on effectue un choix aléatoire.

L'objectif de cette méthode n'est pas de couvrir toutes les transitions globales de la machine composée (graphe d'accessibilité de toutes les *CFSMs*), mais de couvrir toutes les transitions de toutes les *CFSMs*.

Etant donné que pour des systèmes très larges, l'analyse d'accessibilité ne peut être faite manuellement, plusieurs outils automatiques ou semi-automatiques ont été

développés. Nous présentons plus tard dans ce chapitre le langage *SDL* et ensuite l'ensemble d'outils *SDT* qui a été conçu initialement pour valider les protocoles de communication. Ensuite, nous présentons une panoplie d'outils de génération de cas de test dont la plupart sont semi-automatiques et sont basés sur le langage *SDL*.

Le modèle de *FSM* est un modèle très étudié pour la description de machines séquentielles synchrones [Moor 64]. Ce modèle représente tout l'espace d'états explicitement et donc est normalement très large, très cher à construire et inadéquat pour décrire des systèmes complexes. Les modèles *FSM* ont deux inconvénients majeurs: l'incapacité de modéliser correctement la manipulation de variables et l'incapacité de modéliser le transfert de valeurs arbitraires. Dans la section suivante, nous présentons le modèle de *EFSM* qui est une extension du modèle *FSM* et qui permet de modéliser la partie *données* des systèmes.

## 2.7 Machines à états finis étendus

La limitation majeure des *FSMs* est le problème d'explosion d'états ou explosion combinatoire. Afin de décrire un composant d'un protocole qui manipule des compteurs ou bien des structures de données plus complexes, un très grand nombre d'états est requis. Les techniques basées sur le modèle *FSM* sont utiles pour tester l'aspect contrôle d'une *IST*. Cependant, le flux de données d'un protocole est plus facilement représenté sous forme de *FSM* étendue (ou *EFSM*).

Le modèle de *EFSM* décrit un module (processus) comme une *FSM* qui est étendue par ce qui suit:

- Les données d'entrée et de sortie ont des paramètres typés
- Le module a un certain nombre de variables locales typées
- A chaque transition est associé un prédicat, qui dépend des paramètres effectifs de la donnée en entrée reçue et des valeurs courantes des variables locales, et une action qui est réalisée quand la transition est exécutée et qui peut modifier les variables locales.

Une *EFSM* est formellement représentée par un 6-tuple < S, $s_0$, I, O, T, V> où

- S est un ensemble non vide d'états,
- $s_0$ est l'état initial,
- I est un ensemble non vide de messages d'entrée,
- O est un ensemble non vide de messages de sortie,
- T est un ensemble non vide de transitions,
- V est l'ensemble de variables.

Chaque élément de T est un 5-tuple t=(état-source, état-dest, entrée, prédicat, bloc-calcul). Ici, "état-source" et "état-dest" sont des états de S représentant l'état initial et l'état final de t respectivement. "entrée" est soit une donnée d'entrée de I soit l'action nulle. "prédicat" est une condition exprimée en fonction des variables dans V, des paramètres de la donnée d'entrée et de constantes. "bloc-calcul" est un bloc qui constitue les instructions d'affectation et les instructions de sortie de T.

On suppose que l'*EFSM* représentant la spécification est déterministe et complètement spécifiée et que l'état initial est toujours accessible à partir de n'importe quel état dans un contexte valide.

Nous présentons dans les deux sections qui suivent le modèle de système de transitions étiquetées et le modèle des automates à entrées et sorties qui sont aussi des modèles très utilisés pour la spécification des systèmes.

## 2.8 Les systèmes de transitions étiquetées

Un système de transitions étiquetées fini (*LTS* pour Labelled Transition System) est un formalisme utilisé pour décrire des processus.

Formellement, un *LTS* est un quadruplet p = (S, L, T, $s_0$) où:

- S est un ensemble fini non vide d'états
- L est un ensemble fini d'étiquettes
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ est la relation de transition,
- $s_0$ est l'état initial.

Les étiquettes appartenant à L représentent les actions observables du système, alors que l'étiquette $\tau$ représente une action interne non observable. Une trace d'un *LTS* est une séquence d'actions observables du système. L'ensemble de traces d'un *LTS* I est noté par Traces(I) et inclut la séquence vide $\varepsilon$.

Un système de transitions étiquetées est dit non-déterministe lorsque l'une des situations suivantes de produit:

- plusieurs transitions partant du même état sont étiquetées par la même étiquette,
- à partir d'un état, un choix est possible entre une interaction observable et une action interne (non observable),
- le comportement du système en réponse à une sollicitation n'est pas unique. C'est le sens utilisé habituellement dans le contexte du test.

Un opérateur de composition parallèle est défini sur les systèmes de transitions étiquetées.

## 2.9 Les automates à entrées et sorties

Une variante des systèmes à transitions étiquetées est obtenue en subdivisant l'ensemble des actions en deux sous-ensembles, l'ensemble des entrées, c'est à dire les réceptions, et les sorties, c'est à dire les émissions. Cette distinction est due à la nature intrinsèque du test, qui ne peut être mis en oeuvre que comme un dialogue, i.e., une suite d'émissions et de réceptions, entre un testeur et une implantation. Ceci mène à la définition suivante [Phal 94]:

**Définition 4.6:** Un automate à entrées et sorties (*IOSM*: Input-Output State Machine) est un quadruplet (S, L, T, $s_0$) où:

- S est un ensemble non vide d'états
- Lest un ensemble fini non vide d'actions observables
- $T \subseteq S \times (( \{E, R\} \times L) \cup \{\tau\}) \times S$ est la relation de transitions. 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 (E) ou réception (R) d'une interaction), soit une transition interne étiquetée par $\tau$.
- $s_0$ est l'état initial de l'automate.

Dans ce qui suit, nous présentons les réseaux de Petri qui sont un autre formalisme très utilisé pour la description de systèmes.

## 2.10 Les réseaux de Petri

Les réseaux de Petri sont un outil de modélisation graphique et mathématique applicable à plusieurs systèmes. Ils constituent un outil pour décrire et étudier les systèmes de traitement de l'information qui ont la caractéristique d'être concurrents, asynchrones, distribués, parallèles, non-déterministes et/ou stochastiques.

La théorie des réseaux de Petri a été développé dans la dissertation de doctorat du Dr. Petri en 1962. Ces derniers étaient utilisés principalement pour la modélisation des systèmes dont certains événements peuvent survenir d'une manière concurrente. On a aussi découvert qu'ils peuvent être utilisés pour décrire et analyser les systèmes informatiques et des recherches plus récentes suggèrent qu'ils peuvent être utiles dans plusieurs autres domaines tels que: la production, le contrôle automatique, l'analyse de circuits, la physique, l'enseignement, la représentation des connaissances, etc.

La vue d'un système à base de réseaux de Petri se concentre sur deux concepts: les événements et les conditions. Les événements sont des actions pouvant avoir lieu dans le système. L'occurrence de ces événements est contrôlée par l'état du système. L'état du

système peut être décrit par un ensemble de conditions. Une condition est un prédicat ou une description logique de l'état du système.

Un réseau de Petri est un genre particulier de graphe orienté, avec un état initial appelé marquage initial $M_0$. Le graphe N associé à un réseau de Petri est un graphe orienté, bipartie, ayant deux types de noeuds, appelés places (conditions) et transitions (événements). En représentation graphique, les places sont dessinées par des cercles et les transitions par des barres ou des boîtes. Les arcs sont étiquetés par leur poids (entiers positifs), où un arc dont le poids est k peut être interprété comme étant l'ensemble de k arcs parallèles. Un marquage (état) associe à chaque place p un entier non négatif l. On dit que p est marqué par l jetons.

Dans le modèle de réseaux de Petri, deux événements qui sont permis et qui n'interagissent pas l'un avec l'autre peuvent avoir lieu indépendamment. Il n'est pas nécessaire de les synchroniser à moins que ce ne soit requis par le système à modéliser. Quand la synchronisation est nécessaire, elle est facile à modéliser. Les réseaux de Petri peuvent donc être adéquats pour modéliser des systèmes avec un contrôle distribué et ayant plusieurs processus s'exécutant d'une manière concurrente dans le temps. Une autre caractéristique des réseaux de Petri est leur nature asynchrone. Il n'y a pas de mesure de temps ou flux de temps inhérents aux réseaux de Petri. Ceci reflète la philosophie de temps qui dit que la propriété de temps la plus importante, d'un point de vue logique, consiste à définir un ordre partiel pour l'occurrence des événements. Aucune information n'est nécessaire quant à la durée de temps nécessaire à l'exécution de chaque tâche. L'exécution d'un réseau de Petri (et le système comportemental qu'il modélise) est vue comme une séquence d'événements discrets. L'ordre d'occurrence des événements est un parmi plusieurs permis par la structure de base. Ceci entraîne un non-déterminisme apparent dans l'exécution d'un réseau de Petri. Si, à un moment donné, plusieurs transitions sont permises, alors, chacune d'elles peut être la prochaine à être exécutée. Ceci veut dire que le choix de la prochaine transition à exécuter est fait d'une manière non-déterministe ou aléatoire.

Les réseaux de Petri sont une des meilleures méthodes de description de systèmes réactifs complexes. Ils sont graphiques et précis. Aussi, ils viennent avec un ensemble de travaux de recherche, accumulés à travers une trentaine d'années ou plus. Ils sont poussés par les événements et par leur définition même, permettent une concurrence maximale. Cependant, un de leurs inconvénients est la non-disponibilité de décomposition hiérarchique satisfaisante.

La qualité et l'exactitude de tout système est devenue plus significative que jamais. Une erreur de conception peut entraîner la répétition d'un processus coûteux afin de réaliser un nouveau produit. Afin d'éviter de telles dépenses, des méthodologies fiables doivent être développées pour vérifier le processus de conception et obtenir une meilleure confiance ainsi qu'une sécurité dans le produit.

Actuellement, la simulation conventionnelle est la technique la plus utilisée pour vérifier les systèmes matériels ou logiciels et pour détecter les erreurs de comportement assez tôt dans le processus de conception. Cependant, dans les phases finales, la simulation n'est pas efficace et des erreurs peuvent rester cachées. Le problème le plus sérieux est que la simulation ne garantit pas la conformité du système à sa spécification et le niveau de confiance qu'elle offre diminue quand la complexité du système augmente. Ceci est la raison pour laquelle les méthodes de vérification formelle sont développées.

## 2.11 La logique formelle

La vérification formelle est réalisée à l'intérieur d'un système formel, tel que la logique de premier ordre, la logique de haut niveau et la logique temporelle. La phrase "l'implantation I réalise la spécification S" est formulée comme un théorème du système formel. La vérification formelle consiste à offrir la preuve du théorème en question.

Nous devons aussi affronter un problème réel particulier: le temps et les changements temporels. La logique traditionnelle est très puissante face à des situations statiques, mais devient faible quand il s'agit de gérer des situations dynamiques. La vérification temporelle est une autre branche de la logique formelle, applicable à la vérification de systèmes matériels ou à temps réel. En particulier, il y a un intérêt dans la logique

temporelle propositionnelle qui est une extension du calcul propositionnel, adapté à la description comportementale des systèmes qui dépendent du temps. Nous pouvons classer les systèmes de logique temporelle selon la manière avec laquelle ils considèrent le temps. En général, puisqu'il s'agit d'un modèle de temps discret, il existe différentes manières de considérer le futur. Le passé est toujours linéaire alors que le futur peut être un monde unique ou un ensemble de mondes. Dans le premier cas, le temps est aussi linéaire dans le futur, la logique est appelée logique temporelle linéaire et le système possède une évolution unique dans le temps. Dans le deuxième cas, le temps fait des branchements dans le futur, le système est appelé logique temporelle de branchement et le système possède un ensemble d'évolutions possibles.

Dans [Boch 82], un des premiers essais pour utiliser la logique temporelle linéaire comme un formalisme pour décrire et pour faire des raisonnements sur le matériel a été présenté. Dans cette approche, le temps est linéaire et discret. L'auteur a présenté quelques axiomes et a ensuite vérifié un "arbitreur". La vérification est principalement une analyse d'accessibilité où des assertions sont utilisées pour caractériser les ensembles des états possibles du système dans lequel des états et transitions possibles sont générés et symboliquement exécutés. L'auteur a trouvé que la logique temporelle était un outil versatile pour spécifier les propriétés de modules matériels. Aussi, dans [Zhou 87] une logique temporelle pour spécifier le comportement extérieur de systèmes communicants a été introduite. Dans [Bani 87], il a été démontré que la logique temporelle n'était pas suffisante pour spécifier les systèmes communicants ou les systèmes réactifs et qu'une logique temporelle étendue était nécessaire.

Plusieurs langages de spécification sont basés sur le modèle de *EFSM* incluant *SDL*, *Estelle*, *StateCharts* et plusieurs autres langages décrivant le comportement des systèmes orientés objet. La prochaine section introduit quelques uns de ces langages.

## 2.12 Les langages de spécification

Avec le travail sur la normalisation pour l'interconnexion des systèmes ouverts (*OSI*), des groupes travaillant sur les techniques de description formelles ont été établis avec *ISO* et *ITU* au début des années quatre vingt dans le but d'étudier la possibilité d'utiliser des

spécifications formelles pour la définition des protocoles et services de l'*OSI*. Leurs travaux ont abouti à la proposition de langages tels que *SDL* ou *Estelle*. Ces langages sont appelés *techniques de description formelles* ou *TDFs* puisqu'ils possèdent non seulement une syntaxe formelle mais aussi une sémantique formelle qui définit le sens, d'une manière formelle, d'une spécification valide. La sémantique formelle est essentielle pour construire les outils qui vont aider à valider les spécifications et à développer des implantations. Les *TDF*s ont été développées pour assurer:

- des spécifications non ambiguës, claires et concises,
- des spécifications complètes,
- des spécifications consistantes (en isolation et par rapport les unes aux autres),
- des spécifications traçables,
- des implantations conformes aux spécifications.

Nous présentons dans ce qui suit, le langage *SDL* qui est le langage le plus utilisé pour spécifier les protocoles de communication. La section sur *SDL* est plus detaillée que celles pour les autres langages du fait que nous nous intéressons à générer des tests pour des systèmes spécifiés avec ce langage.

### 2.12.1 SDL

*SDL* est le langage de spécification et description recommandé par le *ITU* pour spécifier des systèmes de télécommunication d'une manière non ambiguë. Sa force réside, entre autres, dans le fait que c'est un standard international très reconnu et dans le fait qu'il offre une indépendance des intérêts de certaines compagnies ou vendeurs. *SDL* peut être utilisé pour spécifier d'une manière fonctionnelle le comportement d'objets si ceux-ci peuvent être spécifiés à l'aide d'un modèle discret; i.e., l'objet communique avec son environnement par des messages discrets. Une spécification *SDL* définit le comportement d'un système par le biais de stimuli et réponses. Le modèle de spécification est basé sur le concept de machines à états finis étendus communicantes (*CEFSM*).

Notons que la force de *SDL* et les raisons pour lesquelles plusieurs personnes choisissent de travailler avec ce langage sont les suivantes:

- Il a été conçu spécialement pour décrire les systèmes communicants asynchrones,
- Il est puissant quant à ses capacités d'expression,
- Il est facile à utiliser grâce à son langage graphique,
- Il est spécifié formellement,
- Il est supporté par un ensemble d'outils puissants tels que *SDT* (*SDL* Design Tool) et ObjectGeode.

#### *2.12.1.1 Histoire de SDL*

Le développement de *SDL* a commencé en 1972 et le langage de base était disponible en 1976. Il a été raffiné plus tard en 1980 et la plus grande partie du langage fût définie avant 1984. A la fin de 1988, la définition du langage a atteint une forme stable qui est décrite dans la recommandation Z.100 de *ITU*.

#### *2.12.1.2 Domaines d'application*

De nos jours, *SDL* est principalement connu dans le domaine des télécommunications, mais possède aussi un domaine d'application plus large.

*SDL* peut être utilisé dans les domaines suivants:

- Traitement d'appels dans les systèmes à commutation,
- Maintenance et traitement des fautes dans les systèmes de télécommunication,
- Contrôle de systèmes,
- Protocoles de communication,
- Services de télécommunication.

*SDL* peut être utilisé pour les spécifications fonctionnelles de tout objet dont le comportement peut être spécifié par un modèle discret, i.e., l'objet communique avec son environnement par messages discrets.

*SDL* est un langage riche et peut être utilisé autant pour les spécifications informelles de haut niveau que pour les spécifications semi-formelles ou détaillées. L'usager doit donc choisir les parties de *SDL* appropriées pour le niveau de communication en question ainsi que l'environnement dans lequel le langage est utilisé.

#### *2.12.1.3 Spécification de systèmes*

Dans *SDL*, une spécification est vue comme une séquence de réponses à une séquence de stimuli. Le modèle de la spécification est basé sur le concept de machine à états finis communicantes étendues (*CEFSM*). *SDL* offre aussi des concepts structurés qui facilitent la spécification de systèmes larges ou complexes. Ces constructions permettent de partitionner la spécification en unités pouvant être manipulées et comprises indépendamment. Le partitionnement peut être réalisé en suivant un certain nombre d'étapes résultant en une structure hiérarchique d'unités définissant le système à différents niveaux. La description du système constitue le plus haut niveau de hiérarchie. Un système est ce qui est spécifié par une description *SDL*: c'est une machine abstraite communicant avec son environnement via des canaux. Un bloc est une partie d'un système. Il est aussi composé de processus qui communiquent entre eux et avec les blocs à travers des routes de signaux. Dans la figure 2.5, nous présentons la description d'un système *SDL*.

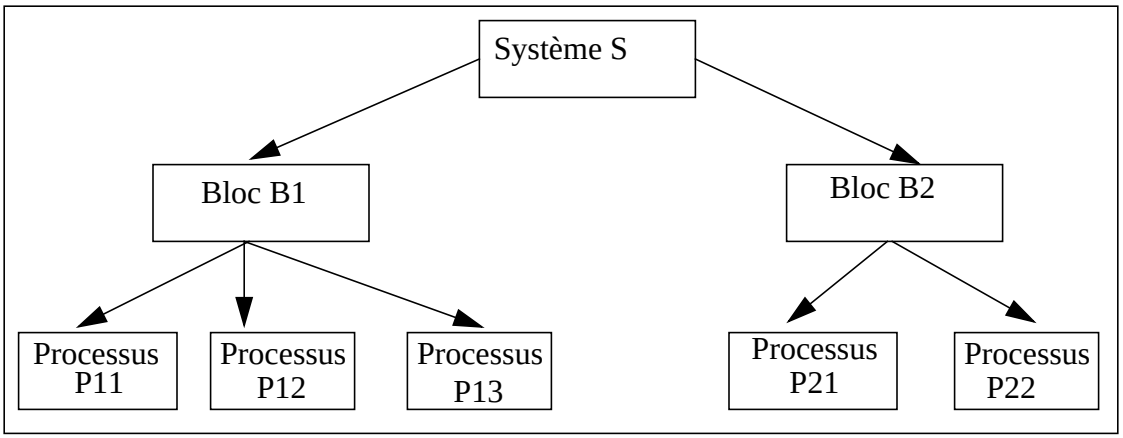



Figure 2.5 Description d'un système en blocs et processus

La figure 2.6 montre le mode de communication entre processus.

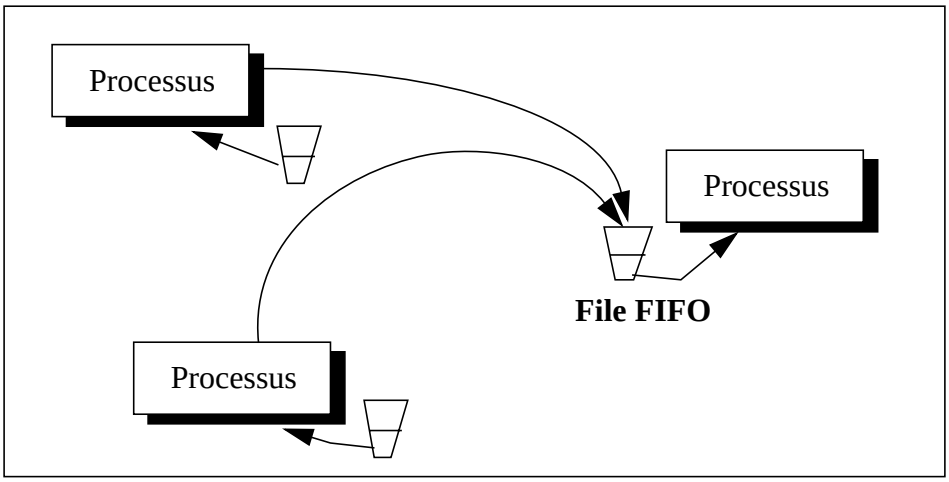



Figure 2.6 Communication entre processus SDL

#### *2.12.1.4 Grammaire de SDL*

*SDL* offre un choix de deux formes syntaxiques à utiliser pour représenter un système: la forme graphique (*SDL/GR*) et la forme textuelle (*SDL/PR*). Comme ces deux formes sont des représentations concrètes du même *SDL*, elles sont équivalentes. Chacune de ces

grammaires possède une définition de sa propre syntaxe ainsi que de sa relation avec la grammaire abstraite.

#### *2.12.1.5 Caractéristiques du language*

La communication est asynchrone, i.e., un message en sortie est stocké dans la file d'entrée du module récepteur avant d'être traité et le processus destinataire d'un message peut être identifié de plusieurs façons, dont les identificateurs de processus ou les noms des canaux ou routes. *SDL* permet le parallélisme, le non-déterminisme et le langage a été étendu pour inclure certaines caractéristiques des spécifications orientées objets.

- Un des avantages à utiliser un langage de spécification formel tel que *SDL* est la possibilité d'effectuer des validations et des vérifications formelles plus précises de la spécification. La spécification peut aussi être utilisée pour dériver les cas de test pour tester l'implantation correspondante à la spécification (pour le test de conformité). Ceci constitue une conséquence directe de la sémantique formelle de *SDL*, puisque celle-ci définit une interprétation non ambiguë et permet un raisonnement formel sur les propriétés de comportement de la spécification.

- Le modèle de concurrence utilisé dans *SDL* suppose que les processus se comportent d'une manière indépendante et asynchrone. Il n'y a pas d'ordre relatif des opérations dans différents processus excepté l'ordre concernant l'envoi et la réception de signaux. Ceci permet aux processus *SDL* d'être implantés d'une manière parallèle sur des unités matérielles séparées, ou bien, d'une manière quasi-parallèle sur du matériel partagé.

- Il existe une file *FIFO* d'entrée associée à chaque processus (premier arrivé premier servi). Tout signal arrivant au processus et qui appartient à son ensemble de données d'entrée valides est mis dans sa file d'entrée. L'ensemble complet des signaux en entrée valides définit les signaux que le processus est prêt à accepter. Les canaux et les routes de signaux représentent les routes de transportation des signaux. Ils peuvent être unidirectionnels ou bi-directionnels. Pour tout signal envoyé par un processus, une et une seule destination doit exister. Celle-ci peut être spécifiée implicitement ou explicitement. Un autre type de communication entre les processus dans *SDL* a lieu lor-

squ'un processus demande la valeur d'une variable appartenant à un autre processus. Ceci est réalisé en utilisant la construction Import/Export ou bien la construction View/ Reveal. Cette dernière ne peut être utilisée que par les processus se trouvant dans le même bloc que la variable à voir. L'opération View n'a cependant pas le droit de modifier la variable.

- La synchronisation dans *SDL* est réalisée en permettant à un processus de vérifier et d'attendre la valeur appropriée d'un signal. La sémantique de *SDL* suppose que l'interaction est asynchrone avec une capacité des files d'attente infinie.

- Une spécification *SDL* contient une hiérarchie d'éléments ou composants qui sont arrangés sous forme d'un arbre. Ceci résulte dans le fait que la spécification possède un certain nombre de niveaux d'abstraction. Le formalisme *SDL* de base permet un seul niveau de concurrence qui consiste en un nombre de machines conventionnelles communicant par un mécanisme de diffusion.

- Chaque processus dans *SDL* possède une adresse unique qui peut être créée par la machine *SDL* sous-jacente durant la création du processus. Les processus peuvent être crées au moment de l'initialisation du système, ou bien plus tard par un autre processus du même bloc. Dans ce cas, toutes les variables de chaque processus sont créées et dans certains cas, elles sont initialisées à des valeurs initiales.

Comparons maintenant *SDL* aux autres langages.

#### *2.12.1.6 Différence entre SDL et les langages de programmation*

- Concurrence: les langages de programmation séquentiels tels que *C* et *Pascal* ne supportent pas la concurrence de *SDL*. Certains langages tels que *CHILL* et *ADA* supportent la concurrence, mais le font différemment.

- Temps: peu de langages de programmation supportent le temps. Le temps tel que défini dans *SDL* n'est pas directement supporté par aucun langage, excepté possiblement par *CHILL* et *VHDL*.

- Communication: la communication des signaux telle que dans *SDL* est supportée par peu de langages.

- Comportement séquentiel: un graphe de processus *SDL* spécifie le comportement transition-état sous forme d'une machine à états finis étendue. Les langages de programmation quant à eux spécifient des séquences d'actions.

- Données: les données de *SDL* sont abstraites et possiblement infinies. L'implantation dans un langage de programmation doit être opérationnelle et finie.

Un tutoriel du langage peut être trouvé dans [Beli 89]. Dans la prochaine section, nous présentons brièvement l'ensemble d'outils *SDT* qui gère et automatise le développement de systèmes en temps réel.

### 2.12.2 Estelle

*Estelle* [Budk 87] a été développé pour la description des services *OSI* et des protocoles. Plus généralement, c'est une technique pour spécifier les systèmes distribués. Le langage est basé sur l'observation qu'un protocole de communication est souvent décrit et implanté à l'aide du modèle d'automate à états finis. *Estelle* se base sur ce modèle en lui ajoutant quelques caractéristiques afin de faciliter l'écriture de descriptions complètes des services de communication et des protocoles. *Estelle* est basé sur le modèle de transition à états finis étendus et sur le langage de programmation *Pascal*. Un système est spécifié comme un ensemble de modules communicants. L'espace d'états d'un module est défini par un ensemble de variables incluant la variable *STATE* qui représente l'état majeur du module. Les interactions d'un module avec son environnement sont spécifiées par des types de transitions. Les transitions sont spécifiées d'un état majeur à un autre état majeur. Elles peuvent dépendre de prédicats sur les variables de contexte comme elles peuvent dépendre de certaines données d'entrée. Les transitions ne dépendant d'aucune donnée d'entrée sont appelées transitions spontanées. La communication entre modules a lieu à travers les points d'interactions d'un module qui a été interconnecté par le module parent. La communication est asynchrone et le modèle *EFSM* permet aux spécifications d'être non-déterministes. Alors que *SDL* a été développé par le *ITU* dès 1972 pour la description

de systèmes de commutation, *Estelle* fût développé par *ISO* pour la spécification de protocoles de communication et de services. Un tutoriel sur *Estelle* peut être trouvé dans [Budk 87]. À la figure 2.7, nous présentons un exemple de transition *Estelle*.

```
from state0
to state1
when data-in(p0,p1)
provided guard
begin
var:=p0;
if var = 10
then q0:=p1;q1:=p1;
else q0:=0; q1:=0;
output data-out(q0,q1)
end;
```

Figure 2.7 Description d'une transition *Estelle*

### 2.12.3 LOTOS

*LOTOS* [Bolo 87] (Language Of Temporal Ordering specification) est utilisé pour modéliser l'ordre d'occurrence des événements d'un système. *LOTOS* possède deux parties clairement séparées. La première offre un modèle comportemental dérivé de l'algèbre des processus, principalement de *CCS* (Calculus of Communicating Systems, [Miln 89]), mais aussi de *CSP* (Communicating Sequential Processes, [Hoar 85]). La deuxième partie du langage permet de décrire des types de données abstraits et des valeurs, et est basée sur le langage de types de données abstraits *ACT ONE* [Ehri 85].

*LOTOS* utilise les concepts de processus, événement et expression comportementale comme concepts de base pour la modélisation.

- Les systèmes et leurs composants sont représentés dans *LOTOS* par des *processus*. Un processus affiche un comportement observable à son environnement en termes de séquences permises d'actions observables. Un processus apparaît comme une boîte noire à son environnement. Les processus communiquent à travers des portes.

- Les spécifications *LOTOS* décrivent les comportements observables des systèmes. Le comportement observable est l'ensemble de toutes les séquences d'interactions dans lesquelles le système peut participer. Ainsi, le concept d'interaction est fondamentale dans le modèle *LOTOS*. Les interactions sont représentées par des *événements*. Ceux-ci sont des interactions atomiques, instantanées et synchrones. A chaque événement est associée une porte (celle où l'événement a lieu). La transmission de données à travers une interface peut être considérée comme un événement. Aussi, le début et la fin de la transmission peuvent aussi être modélisés comme des événements distincts.

- Le comportement observable d'un système *LOTOS* est décrit par une construction du langage dans laquelle les séquences d'événements permises sont définies. Cette construction est appelé *expression comportementale* qui définit les expressions en termes du modèle sémantique de *LOTOS*. Les expressions comportementales peuvent être représentées graphiquement par des arbres de comportement. Cette représentation aide à visualiser la séquence d'événements ainsi que leurs dépendances.

### 2.12.4 StateCharts

Le modèle des *FSMs* est un des formalismes les plus utilisés pour spécifier des systèmes. Deux des inconvénients majeurs de ce modèle sont sa séquentialité inhérente et sa nature non-hiérarchique. Le langage *StateCharts* [Hare 87] a été conçu principalement pour spécifier les systèmes réactifs, qui sont dirigés par les événements et dominés par le contrôle, tels que ceux utilisés en aviation et dans les réseaux de télécommunication. Il dépasse le modèle de *FSM* traditionnel pour inclure trois éléments additionnels: la hiérarchie, la concurrence et la communication.

Le langage *Statecharts* a été proposé comme un formalisme visuel pour décrire les états et les transitions d'une manière modulaire, permettant le groupage, l'orthogonalité (i.e., la concurrence) et le raffinement, et encourageant les capacités de *zoom* pour se déplacer facilement en arrière et en avant entre les niveaux d'abstraction. Techniquement parlant, le noyau de cette approche est l'extension des diagrammes d'états conventionnels par des décompositions d'états en *And / Or* avec les transitions entre les niveaux, et un mécanisme de diffusion pour la communication entre les composants concurrents.

Le mécanisme *StateCharts* a pour but de raviver l'approche naturelle de *FSM* pour la spécification de systèmes, en l'étendant afin de surmonter ses difficultés. Les états dans *StateCharts* peuvent être combinés d'une façon répétitive en états de plus haut niveau utilisant des *AND* et des *OR*. Les transitions dans *StateCharts* ne sont pas restreintes à certains niveaux, et peuvent aller d'un état se trouvant à un certain niveau à un autre. Les données de sortie peuvent être associées aux transitions, comme dans les machines *Mealy*, en écrivant *a/b* sur une flèche; les transitions seront déclenchées par *a* et entraîneront l'événement *b* à se produire. Similairement, *b* peut être associé à un état, comme dans les machines de *Moore*. Dans les deux cas, *b* peut être un événement interne ou externe.

L'usager peut aussi spécifier une borne inférieure et supérieure sur le moment où il faut être à un certain état. *StateCharts* permet la hiérarchie comportementale en permettant à toute spécification d'être décomposée en une hiérarchie d'états. Ceci peut être accompli de deux manières:

- Décomposition *OR* (séquentielle): un état peut être composé d'une machine à états finis comprenant des sous-états séquentiels et des transitions.

- Décomposition *AND* (concurrente): un état peut aussi être composé de sous-états orthogonaux, dans lesquels tous les sous-états deviennent actifs dès que l'état père le devient.

L'inconvénient majeur de *StateCharts* est qu'il sépare le contrôle des données. Ce langage est orienté principalement vers le contrôle et les portions de données sont reliées aux activités des états ou des transitions. Dans [Drus 89], les auteurs ont utilisé *StateCharts* pour la description de matériel et pour la synthèse des systèmes réactifs numériques.

## 2.13 Les outils de support

Dans cette section, nous présentons deux outils de support très connus: *SDT* et ObjectGeode. Ces deux derniers offrent des outils pour aider l'utilisateur à développer des systèmes en permettant d'automatiser et de contrôler le processus de développement.

### 2.13.1 SDT (SDL Design Tool)

*SDT* [SDT 95] offre des outils pour chacune des phases du cycle de développement de systèmes suivantes:

- L'analyse,
- La spécification et la conception,
- La vérification,
- La validation,
- L'implantation et
- La maintenance.

*SDT* est basé sur deux langages définis par *ITU*: le langage *SDL* et le langage pour les *MSC* (qui montre la trace du comportement d'un système) définis dans les recommandations Z.100 et Z.120 du *ITU* respectivement. Les diagrammes de séquences de messages [MSC 92] décrivent des échanges de messages entre les processus *SDL*, et entre processus *SDL* et leur environnement. La partie *implantation* de *SDT* est supportée par les langages C, C++ et CHILL (recommandation Z.200 de *ITU*). Dans ce qui suit, nous présentons d'autres langages qui sont aussi très utilisés pour la spécification des protocoles de communication.

### 2.13.2 ObjectGeode

ObjectGeode [Obje 96] est également un ensemble d'outils qui offre des solutions pour:

- Gérer la complexité en modélisant chaque aspect d'un système (architectural, données, comportemental) d'une manière hiérarchique.

- Utiliser des standards internationaux tels que *MSC* et *SDL*, ainsi que *OMT* (Object Modeling Technique) [Rumb 91] afin de mieux cerner les caractéristiques uniques des systèmes à temps réel. *MSC* permet de décrire les scénarios représentant les interactions entre les différents composants du système. *SDL* permet de concevoir, graphiquement, des systèmes réactifs à travers le modèle *EFSM* et *OMT* permet de décrire les données d'une manière orientée objet.

- Produire des systèmes de haute qualité en appliquant les techniques de vérification et de validation tôt dans le cycle de développement.

- Livrer des systèmes prêt à être exécutés dans des environnements à temps réel.

- Automatiser et contrôler tout le processus de développement, i.e., de l'analyse à la génération de code, au test et à la maintenance.

## 2.14 Conclusion

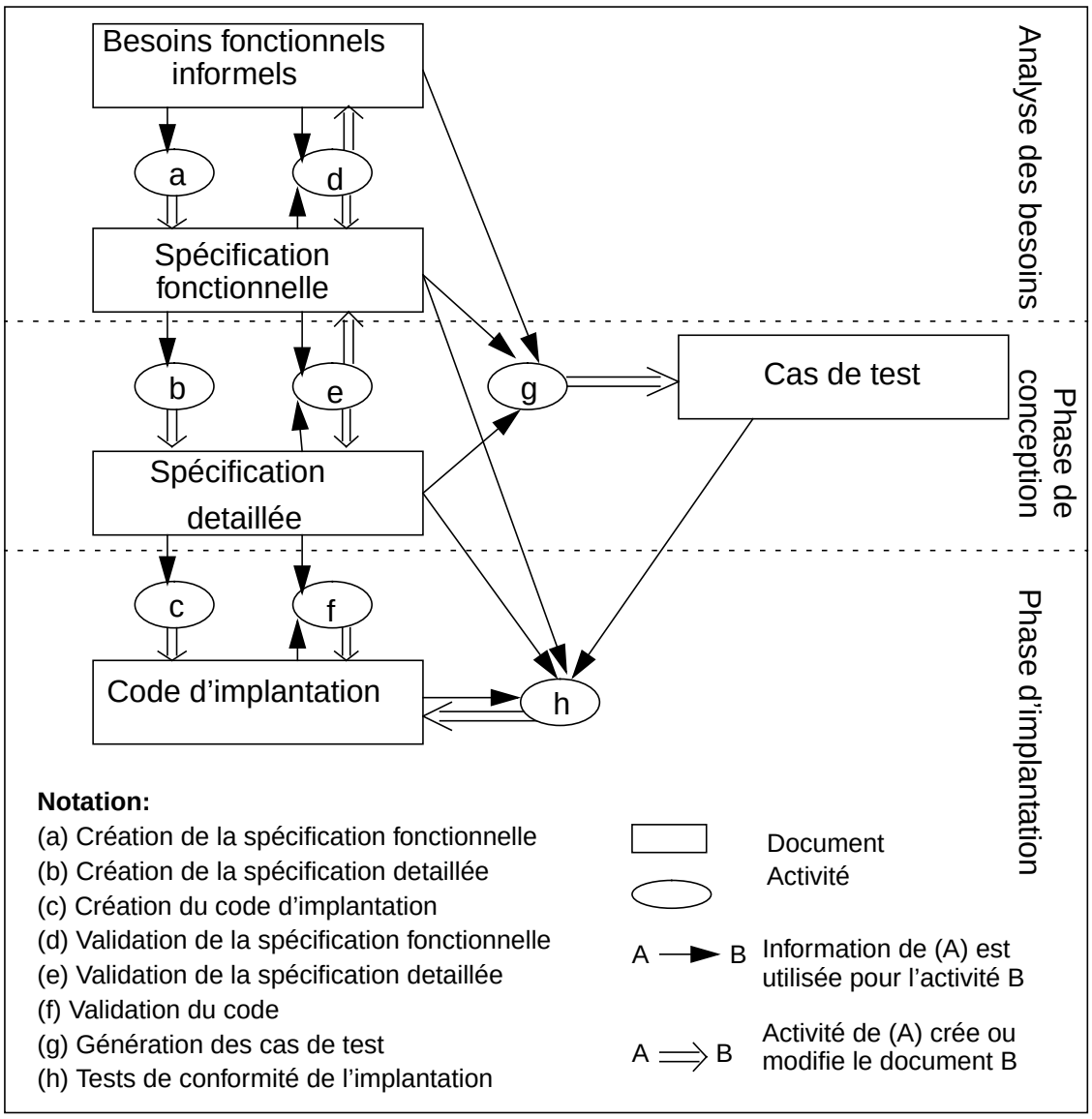



Figure 2.8 Les activités de développement de systèmes

Comme le montre la figure 2.8 [Boch 90], les spécifications fonctionnelle et détaillée sont utilisées dans la plupart des activités de développement durant le cycle de développement d'un système. Ainsi, les techniques de description et les langages utilisés pour ces spécifications ont aussi un impact majeur sur le cycle de développement du système. Plusieurs raisons encouragent l'utilisation des méthodes formelles pour la

spécification du comportement d'un protocole. La raison principale de l'utilisation des techniques de description formelles est l'usage d'outils mathématiques de vérification ainsi que le fait que certaines activités du cycle de développement deviennent automatisables.

L'étape de spécification est l'une des plus importantes du cycle de développement. Les erreurs introduites au niveau de la spécification se répercutent automatiquement sur le produit final. Depuis quelques années les spécifications sont de plus en plus décrites par des langages formels et normalisés appelés *Techniques de Description Formelle TDFs* (*FDT*: *Formal Description Techniques*): par exemple, *Estelle* [Budk 86], *LOTOS* [Bolo 87], *SDL* [Beli 89], Z [Spiv 92].

Les prochains points indiquent comment ces activités peuvent bénéficier de l'utilisation des *TDF*s.

- Validation de la spécification: pour la validation de la spécification par les tests, les approches décrites dans les deux derniers points peuvent être appliquées si la spécification est exécutable.

- Raffinement de la spécification et de l'implantation: la spécification, orientée implantation, peut être développée à partir de la spécification d'origine. La spécification raffinée peut être utilisée pour la génération semi-automatique du code.

- Sélection des cas de test: des méthodes partiellement automatiques existent pour la sélection des cas de test basés sur la spécification formelle du système à tester.

- Analyse des résultats de test: les résultats des tests réalisés sur l'implantation sous test (*IST)* doivent être analysés et comparés à ceux générés à partir de la spécification et dans le cas des spécifications formelles, cette comparaison peut être automatisable.

Les *TDF*s ont été introduites non seulement pour réduire l'ambiguïté engendrée par les langages naturels, mais aussi pour automatiser le plus possible chacune des étapes du cycle de développement. Cependant, même à partir d'une spécification formelle, la production d'une implantation demeure une tâche délicate. En effet, une spécification est

une description abstraite, elle décrit ce que le système est supposé accomplir, sans fixer comment il doit le faire. Cette description abstraite peut être réalisée de plusieurs manières, en utilisant différents langages de programmation et différentes architectures. Le processus de raffinement d'une spécification pour l'obtention d'une spécification moins abstraite puis d'une implantation est complexe et est sujet à différents types d'erreurs.

L'ultime recours pour s'assurer qu'un produit fait bien ce que l'on attend de lui est de le tester. L'effort de test représente une part significative du coût de développement d'un système. L'activité de formalisation du test fait l'objet de travaux de recherche depuis une bonne trentaine d'années. Cette activité utilise principalement les techniques de test de logiciel ainsi que celles du modèle de machines à états finis. Pour tester un système, il faut 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, appelé aussi implantation, est conforme à sa spécification.

# Chapitre 3

# Le test de logiciel

## 3.1 Introduction

Dans l'industrie, le test joue un rôle très important. C'est une activité associée à tout processus qui génère un produit. Elle peut être simple mais plus souvent, c'est un ensemble d'étapes qui vérifient si le produit va être commercialisé ou non.

L'étape de test représente la dernière activité du cycle; elle suit immédiatement celle d'implantation. Elle permet de vérifier l'implantation avant de délivrer le produit. L'étape de test est donc l'étape qui garantit une certaine qualité d'une implantation. Elle consiste à valider le comportement de l'implantation par rapport à des données d'entrées spécifiques (cas de test) sélectionnées à l'avance. L'objectif de cette étape est alors la stimulation des fautes, l'observation de leurs manifestations (erreurs), leur localisation et éventuellement leur correction. Dans la littérature, les deux dernières activités (localisation et correction des erreurs) peuvent être regroupées dans une étape indépendante appelée débogage ou *diagnostic* [Myer 79].

## 3.2 Définition des tests

Il n'y a pas de définition de test sur laquelle tout le monde est d'accord. Le terme est souvent utilisé pour décrire des techniques qui vérifient un logiciel en l'exécutant avec des données. Le test peut aussi avoir un sens plus large: le test inclut toute technique de vérification de logiciel, telle que l'exécution symbolique et la preuve de programmes ainsi que l'exécution d'un programme avec des données. La vérification implique qu'une

comparaison est réalisée. La comparaison a lieu entre les sorties du test et les sorties attendues dérivées par le testeur. Les sorties attendues sont extraites de la spécification.

Plusieurs recherches sur les tests ont été réalisées. On y trouve une multitude de définitions plus ou moins différentes. Ci-dessous, nous en présentons quelques unes.

- *Le processus de test consiste en une collection d'activités qui visent à démontrer la conformité d'un programme par rapport à sa spécification. Ces activités se basent sur une sélection systématique des cas de test et l'exécution des segments et des chemins du programme* [Wass 77].

- *Les tests représentent l'essai d'un programme dans son milieu naturel. Ils nécessitent une sélection de données de test à soumettre au programme. Les résultats des traitements de ces données sont alors analysés et confirmés. Si on découvre un résultat erroné, l'étape de débogage commence* [Clar 76].

- *L'objectif du processus de test est limité à la détection d'éventuelles erreurs d'un programme. Tous les efforts de localisation et de correction d'erreurs sont classés comme des tâches de débogage. Ces tâches dépassent l'objectif des tests. Le processus de test est donc une activité de détection d'erreurs, tandis que le débogage est une activité plus difficile consistant en la localisation et la correction des erreurs détectées* [Whit 78].

## 3.3 Objectifs du test

Dans son livre sur le test de logiciel, Glen Myers [Myer 79] énonce un certain nombre de règles qui peuvent servir comme objectifs de test.

- *Un bon cas de test est celui qui a une forte probabilité de trouver une erreur non encore découverte.*

- *Un test réussi est celui qui découvre une erreur non encore découverte.*

L'objectif est de concevoir des tests qui découvrent systématiquement différentes classes d'erreurs avec un minimum de temps et d'effort. Selon cet objectif, si le test est

conduit d'une manière réussie, il va découvrir des erreurs dans le logiciel. Un autre avantage est que le test démontre que les fonctions d'un logiciel semblent marcher selon la spécification et que les besoins de performance semblent être rencontrés (satisfaits). Mais il y a une chose que le test ne peut pas faire: le test ne peut pas montrer l'absence de fautes, il peut seulement montrer que certaines fautes sont présentes.

## 3.4 Définition des fautes, des erreurs et des défaillances

Les notions de fautes, d'erreurs et de défaillances du matériel sont fortement liées. Leur définition a été adaptée du matériel [Arse 80, Prad 86, Lapr 91], etc:

- La faute représente une condition anormale dont la manifestation est une erreur. Elle consiste en un état logique du système différent de celui attendu.
- L'erreur est la conséquence directe ou indirecte d'une faute, elle en est la manifestation.
- La défaillance est l'effet d'une erreur sur le service, elle survient lorsque le système a un comportement erroné.

Les tests visent à stimuler les fautes, observer leur manifestation, localiser les erreurs et éventuellement les corriger. Les tests réduisent la probabilité que des utilisateurs potentiels du produit puissent y observer une défaillance. Les fautes peuvent être classées selon leur durée, leur origine, leur nature et leur étendue. Du point de vue durée, une faute peut être transitoire, intermittente ou permanente. Une faute commise devient une erreur latente aussi longtemps qu'elle n'a pas été activée. La nature de la faute dépend de son comportement. L'étendue de la faute peut être locale en affectant un seul module, ou globale en affectant plusieurs modules interconnectés. L'origine de la faute est soit humaine soit physique. Les fautes peuvent être simples ou multiples. En général, la présence ou la considération de fautes multiples rend les processus de test et de diagnostic plus complexes.

## 3.5 Les principales stratégies de test

Historiquement, plusieurs classifications de techniques de tests ont émergé. Certains

articles font la distinction entre les techniques de test statiques et dynamiques. Un autre ensemble de termes qui nécessite une explication est la dichotomie test de boîte blanche/ noire (test structurel/fonctionnel).

### 3.5.1 Test fonctionnel versus test structurel

Mettre le code à la disposition du test ou non va définir les stratégies de test. On distingue alors le test fonctionnel du test structurel. Dans la première stratégie de test, les fonctions nécessaires sont identifiées à partir de la spécification. Par la suite, le logiciel est testé en vue de vérifier si ces dernières sont offertes. Par contre, la deuxième stratégie est basée sur la dérivation des données de test à partir de la structure du système.

Test structurel

> Ce genre de test est aussi appelé test de *boîte blanche*. Plutôt que d'être fondé sur les fonctions du logiciel, le test structurel se base sur le détail du logiciel (d'où son nom). Les tests structurels sont effectués sur des produits dont la structure interne est accessible et sont dérivés en examinant l'implantation du système (style de programmation, méthode de contrôle, langage source, conception de base de données, etc). Ils s'intéressent principalement aux structures de contrôle et aux détails procéduraux. Ils permettent de vérifier si les aspects intérieurs de l'implantation ne contiennent pas d'erreurs de logique. Ils vérifient si toutes les instructions de l'implantation sont exécutables (contrôlabilité). Le système à tester est à comparer à une spécification de référence plus abstraite. Cependant, les connaissances sur le système à tester ainsi que sur la spécification de référence sont utilisées pour la sélection des données de test, des critères de couverture ainsi que pour l'analyse des résultats du test.

Test fonctionnel

Le test fonctionnel s'intéresse aux besoins fonctionnels du logiciel (il est aussi appelé test de *boîte noire*). Ce type de test comporte deux étapes importantes: l'identification des fonctions que le logiciel est supposé offrir et la création de données de test qui vont servir à vérifier si ces fonctions sont réalisées par le logiciel ou pas.

Les cas de test qui sont dérivés sans référence à l'implantation du système (i.e., sont crées par référence à la spécification, ou à une description de ce que le système est supposé faire) sont appelés tests de boîte noire. Ces tests ne sont pas une alternative aux tests de boîte blanche. Il constituent plutôt une approche complémentaire qui découvre des classes différentes d'erreurs. Dans cette approche, le système est traité comme une boîte noire et sa fonctionnalité est déterminée en injectant au système différentes combinaisons de données d'entrées. L'objectif de ces tests est la vérification de la conformité des fonctionnalités de l'implantation par rapport à une spécification de référence. Les détails internes des logiciels ne sont pas inspectés. Ils ne s'intéressent qu'au comportement externe du système à tester. Ces tests permettent de détecter une large gamme d'erreurs comme: fonctions incorrectes ou manquantes, erreurs d'interfaces, erreurs de structures de données ou d'accès aux bases de données externes, problèmes de performance, erreurs d'initialisation ou de terminaison. Les tests de boîte noire sont généralement réalisés durant les dernières phases du test.

Mis à part ces deux stratégies, il existe une stratégie hybride appelée *test boîte grise* qui mérite d'être définie.

Tests boîte grise

Parfois, le terme "test de boîte grise" est utilisé pour dénoter la situation où la structure modulaire du produit logiciel à tester est connue des testeurs, c'est-à-dire lorsqu'on peut observer les interactions entre les modules qui composent le logiciel, mais pas les détails des programmes de chaque composante. L'observation des interactions se fait sur des points spécifiques du logiciel appelés points d'observation. Cette approche n'a donc pas besoin des détails internes des modules,

mais exige l'observabilité de la structure du logiciel via les point d'observation. La visibilité de la structure interne du logiciel peut énormément aider lors de l'étape de débogage.

### 3.5.2 Analyse statique versus analyse dynamique

Une technique de test qui ne comprend pas l'exécution du logiciel avec des données est appelée *analyse statique*. Ceci inclut la *preuve de programme*, l'exécution *symbolique* et l'*analyse d'anomalies*. La preuve de programme inclut la spécification rigoureuse de contraintes sur les données d'entrée et sur les données de sortie pour un composant du logiciel. La portion de code qui implante ce composant est alors prouvée mathématiquement. L'exécution symbolique est une technique qui exécute un système logiciel, avec des valeurs symboliques des variables utilisées, plutôt qu'avec les valeurs numériques. L'analyse d'anomalies est une technique qui vérifie un programme afin de trouver des caractéristiques présentant des anomalies telles qu'un code isolé.

L'*analyse dynamique* requiert que le logiciel soit exécuté. Elle repose sur l'utilisation de *requêtes* insérées dans le programme. Ces dernières sont des instructions du programme qui font des appels à des routines d'analyse qui enregistrent la fréquence d'exécution de certains éléments du programme. Par la suite, le testeur est capable d'affirmer certaines informations telles que la fréquence d'exécution de certaines branches ou instructions du programme et si certaines régions du code ne sont pas touchées par le test.

L'analyse dynamique peut agir comme un pont entre le test fonctionnel et structurel. Initialement, le test fonctionnel peut montrer l'ensemble des cas de test. L'exécution de ces cas de test peut être surveillée par l'analyse dynamique. Le programme peut alors être examiné d'une manière structurelle pour déterminer les cas de test qui seront utilisés pour exécuter le code n'ayant pas pu être exercé par les cas de test précédents. Cette approche duale assure que le programme est testé pour les fonctions requises et que tout le programme est testé par les cas de test.

### 3.5.3 Test ascendant versus test descendant

Tests ascendants

C’est une stratégie classique de test. Elle consiste à tester les modules, ensuite les sous-systèmes et en dernier le système en entier. La première phase consiste à tester les modules dans l’objectif de découvrir des erreurs de logique, de fonctionnalité ou de structure. Les modules sont testés individuellement dans un environnement qui simule le sous-système qui les englobe. Par la suite on passe aux tests des sous-systèmes en vérifiant les interactions entre les différents modules du sous-système. Ils s’intéressent principalement aux interfaces des modules et aux échanges entre ces derniers. Le test des sous-systèmes est accompli par un processus répétitif qui permet d’intégrer les modules de base aux sous-systèmes, les sous-systèmes d’un certain niveau aux sous-systèmes d’un niveau plus haut et ainsi de suite jusqu’à tester tout le système. Ces tests sont donc hiérarchiques et doivent se faire du bas vers le haut. Les techniques de test ascendants utilisent plusieurs types de test comme: les tests unitaires pour tester les modules et les tests d’intégration pour tester les sous-systèmes.

Tests descendants

Cette approche est moins naturelle que la précédente. Elle commence par tester le programme principal avec ses sous-routines immédiates. Après cela, on passe aux tests de plus bas niveaux. Ces derniers se basent sur ce qui a été déjà validé pour tester les modules d’un niveau plus bas. Ces techniques ont besoin d’utiliser des modules factices qui seront placés à un niveau plus haut que les modules à tester et qui ont pour rôle de récolter les sorties des modules à tester. Cette stratégie suppose que la structure du système est hiérarchique. Dans plusieurs cas, elle est impossible à effectuer, car les données d’entrées qui servent à tester un module proviennent des modules appartenant à des niveaux supérieurs. Le problème est de trouver les entrées adéquates des modules de niveaux supérieurs qui produisent les entrées de test pour les modules en question.

## 3.6 Les principaux types de tests de logiciels

### 3.6.1 Tests unitaires

Ces tests visent à vérifier individuellement chaque module [Myer 79]. Ils permettent de tester l'interface et l'intérieur du module soit: les structures de données locales, les chemins, les instructions, les conditions, les boucles, les entrées sorties et les traitements en cas d'erreurs. Ces tests sont effectués par l'implanteur même, puisqu'ils nécessitent la disponibilité du code et les détails internes du logiciel au complet. C'est pour cette raison que le test unitaire est normalement orienté test de boîte blanche et cette étape peut être conduite en parallèle pour plusieurs modules.

### 3.6.2 Tests d'intégration

Le test d'intégration est une technique systématique pour construire la structure d'un programme tout en conduisant des tests qui découvrent des erreurs associées à l'interfaçage. L'objectif est de prendre les modules testés individuellement (à l'aide du test unitaire) et de construire une structure de programme selon les besoins de conception. Après que les modules aient été testés individuellement, on vérifie leur comportement ensemble, c'est-à-dire on les intègre ensemble et on teste si leur interaction est conforme à la spécification de référence. L'intégration des modules se fait de plusieurs façons: incrémentale ascendante, ou descendante, ou hybride ascendante et descendante (*sandwich*), ou intégration du tout à la fois. Ces tests sont moins exigeants que les tests unitaires et ne nécessitent que la connaissance de la structure modulaire et des interfaces des modules du logiciel.

### 3.6.3 Test de conformité

Le test de conformité [Rayn 87] des protocoles est une approche systématique utilisée pour s'assurer que les produits vont se comporter comme prévu [ISO 94A, 94B, 92, 94C, 94D, 94E]. Plus spécifiquement, le test de conformité joue un rôle très important dans la vérification de la conformité des produits aux standards *ISO*, *ITU*, aux recommandations et aux spécifications du vendeur.

Le test de conformité est une approche de boîte noire si un testeur externe peut seulement observer les sorties générées par l'implantation suite à la réception de données d'entrées (à l'inverse du testeur qui a une connaissance de la structure interne de l'implantation).

La conformité est une relation, parmi plusieurs, entre la spécification et l'implantation. Les autres types de relation peuvent être trouvées, i.e., les relations d'équivalence, dans [Ledu 91]. La relation de conformité est valide si l'implantation ne présente pas un comportement non acceptable par la spécification. Si l'implantation est une boîte noire, on ne peut tester que son comportement observable contre celui de la spécification. Durant le test de conformité, des signaux sont envoyés à (données d'entrée) et reçus de (données de sortie) l' implantation. Les signaux de l'implantation sont comparés aux signaux attendus de la spécification. Les données d'entrée et les données de sortie attendues sont décrites dans ce que l'on appelle une suite de test. Une suite de test est structurée en cas de test. L'exécution d'un cas de test donne un verdict. Des verdicts, une conclusion sur le résultat de conformité est déduite. Un verdict peut être "pass", "échec" ou "inconclusif".

- Un verdict "pass" est donné quand les sorties observées satisfont le but de test et lorsqu'elles sont complètement valides par rapport aux besoins de conformité.

- Le verdict "échec" veut dire que les sorties observées sont invalides par rapport aux besoins de conformité;

- Le verdict "inconclusif" est donné quand les sorties observées sont valides par rapport aux besoins de conformité et invalides par rapport au but de test.

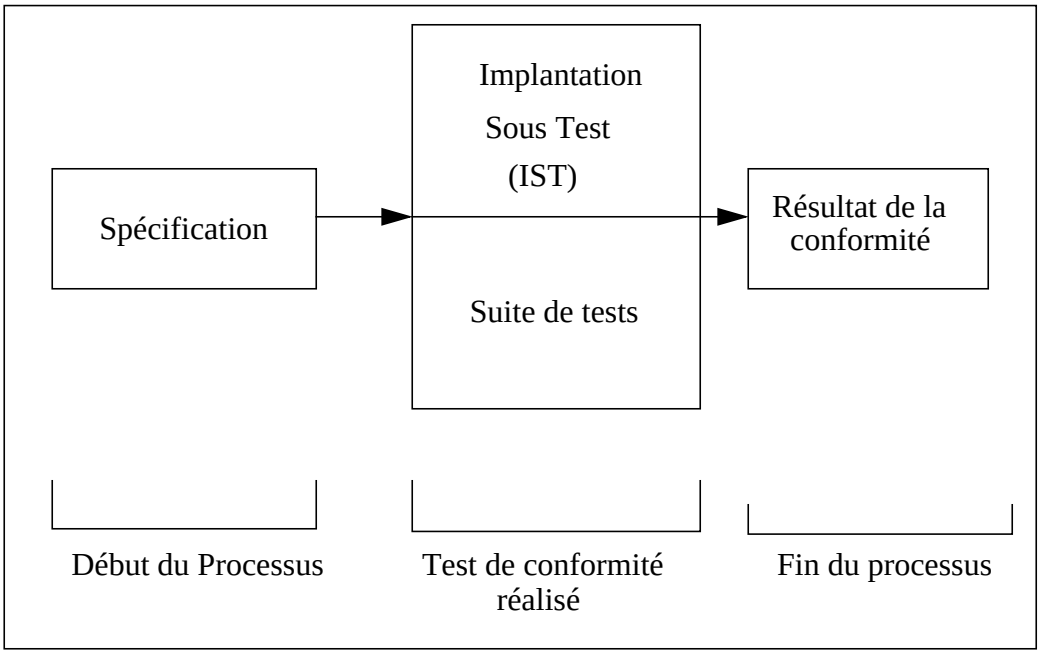



Figure 3.1 Processus du test de conformité

Le test de conformité comprend trois étapes principales:

1. Spécifier la suite de test: offrir une procédure de développement des suites de test finies pour une certaine spécification.
2. Appliquer la suite de test à l'implantation.
3. Analyser les résultats de test en évaluant la relation de conformité après l'application de la suite de tests.

A un niveau d'abstraction élevé, l'architecture de test consiste en un testeur, une implantation (*IST*) sous test et un contexte de test (voir figure 3.2). Le testeur envoit et reçoit des signaux de l'*IST* et ne peut accéder à l'*IST* que via le contexte de test. La spécification est donnée sous forme de boîte blanche et offre une définition des traces valides de l'architecture de test. L'implémentation est donnée sous forme de boîte noire et peut être évaluée. Le testeur peut seulement observer et contrôler un sous-ensemble des événements exécutés (ceux qui envoient et qui reçoivent des signaux aux *PCOs*: i.e, les événements observables).

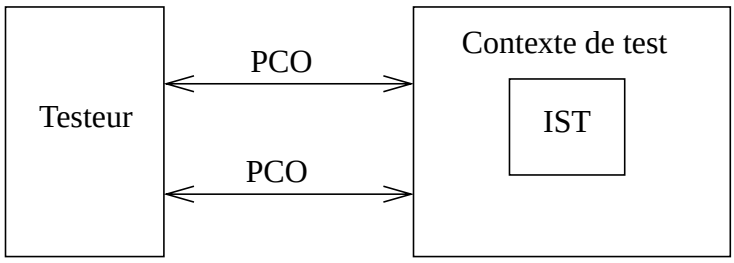



Figure 3.2 Architecture de test abstraite

Un test de conformité vise à vérifier si une implantation donnée d'un protocole peut être considérée équivalente à sa spécification. Le test ne peut jamais garantir l'absence d'erreurs et donc ne peut assurer une fiabilité complète. Seul le comportement désiré peut être testé et pas l'absence de comportement indésirable. Il est donc possible qu'une implantation soit capable de donner des réponses qui ne font pas partie de la spécification. Ainsi, même si le test augmente la probabilité de l'interoperabilité d'un système, il ne peut jamais garantir une très forte conformité (au sens de l'équivalence) à toutes les spécifications possibles. Le test de conformité essaie de vérifier le comportement d'une implantation sujette à des conditions ou erreurs spécifiques. Le groupe de conditions choisies par le concepteur de test comme critères de conformité comprend ce que l'on appelle les besoins de conformité.

Ce genre de tests occupe une place très importante dans le domaine de l'ingénierie du protocole [Raie 87]. Il consiste à vérifier la conformité de l'implantation par rapport à la spécification de référence. Ces tests se basent sur le comportement extérieur de l'implantation. Ils consistent à appliquer un ensemble d'entrées spécifiques (cas de test) au système à tester et à vérifier la conformité de ses sorties. Généralement, un tel test ne peut pas être exhaustif vu le nombre important de cas à traiter. Les entrées sont alors sélectionnées de façon à exécuter des parties du système (fonctions, transitions, états, etc) et à détecter le maximum de fautes.

Nous tenons aussi à définir plus en détail la vérification et la validation afin de ne pas les confondre avec le test de conformité.

### 3.6.4 Vérification

La vérification consiste à établir l'exactitude d'une spécification et/ou implantation d'un système. En d'autres mots, l'objectif de la vérification est de répondre à la question suivante: "Est-ce que le système est exact?". Les techniques de vérification sont applicables si la structure interne de l'implantation est connue. La vérification formelle est utilisée quand le système est bien décrit dans un cadre formel, tel que la logique de premier ordre, la logique d'ordre supérieur ou la logique temporelle.

Il existe deux méthodes de vérification: la vérification de modèles (model checking) et la preuve de théorème.

- La vérification de modèle [Clar 86, Gode 93, Wolp 89] est une approche orientée état. La spécification et l'implantation sont décrites à l'aide d'une machine à états. En explorant tous les états de l'implantation et de la spécification, on vérifie si l'implantation peut exécuter une trace non décrite par la spécification. L'avantage de cette méthode est l'existence d'algorithmes qui font la vérification de modèles automatiquement. Par contre, cette méthode n'est applicable que pour les modèles à états finis. De plus, elle possede plusieurs limitations quant au nombre d'états de la specification et de l'implantation.

- La preuve de théorème [Kroe 86, Mann 83, Mann 90] est une technique orientée événements. L'implantation et la spécification doivent être formalisées selon un calcul logique, i.e., l'effet des actions est spécifié à l'aide de formules logiques ou temporelles. La preuve de théorème montre que la formule logique de l'implantation est une conclusion de la formule logique de la spécification. "Est-ce que l'implantation I est conforme à la spécification S?" est formulée comme un théorème du système formel. La preuve de théorème consiste à offrir la preuve du théorème en question.

### 3.6.5 Validation

Afin de valider les protocoles, il faut s'assurer que les propriétés suivantes sont vérifiées:

- La sûreté (absence d'interblocages, de réception non spécifiée, de cycles bloquants, etc, ces termes seront définis plus loin)

- La vivacité: un état (une transition) est vivant(e) s'il (elle) peut être atteint(e) à partir de tous les états du système global. Normalement, un protocole contient un état initial à partir duquel toutes les opérations pertinentes peuvent être atteintes.

La méthode de validation des protocoles la plus populaire et la plus utilisée est l'analyse d'accessibilité. Elle part d'automates finis décrivant des machines communicantes échangeant des messages à travers des canaux (files d'attente *FIFO*), pour développer un graphe de communication global appelé graphe d'accessibilité. Les états et la structures de ce graphe sont ensuite examinés à la lumière d'un certain nombre de propriétés pour valider la communication mise en oeuvre.

Cette méthode est utilisée depuis longtemps. Cependant, elle possède deux inconvénients majeurs:

- l'indécidabilité de la finitude du graphe d'accessibilité dans le cas de canaux non bornés,

- l'explosion combinatoire de ce graphe dans le cas fini.

Dans la pratique, i.e., dans les réseaux informatiques, le premier problème ne se pose pas car la taille des canaux de communication est finie. Le deuxième inconvénient est en fait un problème inhérent à toutes les méthodes de preuve et de validation par développement du graphe de tous les cas possibles.

Il est difficile de définir la validation des protocoles. Chacun des points suivants décrit un certain aspect du fonctionnement d'un protocole et la validation des protocoles peut être considérée comme étant l'analyse des ces différents aspects et la comparaison des résultats obtenus avec les besoins de fonctionnement.

- Analyse d'accessibilité: la base de tous les aspects de la validation est une analyse des transitions possibles du système global, qui est le produit cartésien des espaces d'états des composants du système. L'analyse d'accessibilité produit le diagramme de transitions du système global. Un algorithme d'analyse d'accessibilité essaie de générer et de vérifier tous les états d'un système distribué qui sont accessibles à partir d'un certain état initial. Implicitement, il construit toutes les séquences d'exécution possibles. Il y a trois types d'algorithmes d'analyse d'accessibilité: la recherche complète pour des systèmes dont le nombre d'états peut atteindre $10^5$ états, la recherche partielle contrôlée pour des systèmes de $10^8$ états et la simulation aléatoire pour les systèmes plus larges.

- Les interblocages: un interblocage est caractérisé par un état du système global, accessible à partir de l'état initial et à partir duquel aucune transition n'est possible.

- La vivacité (définie plus haut)

- Les boucles: chaque protocole ayant un état initial peut contenir une boucle dans le diagramme de transitions du système global commençant et finissant à cet état. D'habitude, il peut y avoir d'autres boucles nécessaires à son fonctionnement. De plus, durant la phase de conception d'un nouveau protocole, d'autres boucles peuvent être introduites dans le diagramme de transitions. Ces dernières forment un comportement indésirable.

- L'auto-synchronisation et la stabilité: un système est auto-synchrone si, lorsque démarré à n'importe quel état possible du système, il retourne toujours, après un nombre fini de transitions, au cycle normal de fonctionnement incluant l'état initial. Cette propriété est importante pour la correction des anomalies ou des comportements non prévus pouvant survenir dans un environnement non fiable. La propriété d'auto-synchronisation implique la stabilité du protocole, du fait qu'elle assure que le protocole revient directement à son mode de fonctionnement normal après une perturbation dans la synchronisation de ses sous-systèmes communicants.

### 3.6.6 Tests de l'usager

Ce sont les tests effectués au niveau de l'usager en vue de vérifier si le produit final répond bien à ses besoins. Généralement, ces tests consistent à essayer le système sur des situations réelles et propres à l'environnement de l'usager. Le résultat de ces tests est l'acceptation, l'acceptation conditionnelle ou le refus du produit.

### 3.6.7 Tests d'interopérabilité

Ce type de test évalue le degré d'interopérabilité de plusieurs implantations. Il implique le test des capacités ainsi que du comportement de l'implantation dans un environnement inter-connecté. Il permet aussi de vérifier si une implantation est capable de communiquer avec une autre implantation de même type ou de type différent. Les deux approches de test d'interopérabilité dans le contexte de l'*OSI* sont le test actif et le test passif [Gadr 90, Hogr 90]. Le test actif permet la génération d'erreurs contrôlées et une observation plus détaillée de la communication, alors que le test passif implique le test du comportement valide seulement.

### 3.6.8 Tests de performance

L'objectif est de tester la performance d'un composant du réseau face à des situations normales et parfois extrêmes [Schi 97]. Le test de performance identifie les niveaux de performance du composant du réseau pour différentes valeurs des paramètres et mesure la performance du composant. Une suite de tests de performance décrit précisément les caractéristiques de performance qui ont été mesurées ainsi que les procédures montrant la manière d'exécuter ces mesures. Ce genre de tests est très important pour les systèmes à temps réel.

### 3.6.9 Tests de robustesse

Ces tests s'intéressent au degré de résistance de l'implantation à des événements externes ou à des erreurs non prévues par la spécification. C'est-à-dire les capacités du système à fonctionner même dans des situations non prévues par la spécification de référence.

Selon ce qu'on veut tester dans un système (la logique, la structure, les fonctions, etc), les tests varieront et seront orientés parfois au code, parfois à l'architecture et parfois aux entrées-sorties. Les tests de conformité par exemple ne s'intéressent qu'aux entrées-sorties du système, ces tests sont dits tests de *boîte noire*. D'un autre côté, les tests d'intégration s'intéressent plutôt à l'architecture et à la structure modulaire du système, ces tests sont dits tests de *boîte grise*. Finalement, les tests unitaires se basent sur le code de chaque module pour mener à bien leur vérification, ce genre de tests sont appelés les tests de *boîte blanche* [Myer 76].

## 3.7 Les modèles de fautes

Les tests ont pour principal objectif la détection d'erreurs d'une implantation. Une erreur est la conséquence directe ou indirecte de l'exécution d'une faute (défectuosité dans le système). Pour tester le système, on doit donc tenir compte des différentes fautes possibles. Ces dernières peuvent être nombreuses et complexes. De plus, elles peuvent engendrer la même manifestation (erreur). Ceci nous mène à classer les fautes d'après les erreurs qu'elles induisent. Elles seront alors réunies au sein de modèles différents appelés modèles de fautes [Boch 91d]. Ces derniers permettent de décrire l'effet des fautes à un niveau d'abstraction élevé. Ceci aura pour avantage de réduire le nombre de possibilités à traiter lors de la génération de cas de test.

Les fautes communes à tous les logiciels peuvent être classées en deux catégories principales:

- les fautes de traitement: fautes de séquencement dans le programme, fautes d'opérations arithmétiques et de manipulation de données, fautes dans l'appel de fonctions, etc.

- les fautes de données: fautes de type ou de représentation du format de données, fautes d'initialisation ou du domaine d'une donnée, référence à une variable qui n'est pas la bonne, fautes de référence de variables indéfinies, fautes de définition de variables non utilisées.

Malgré que l'ensemble des fautes ne soit pas complètement connu et que les modèles proposés ne couvrent qu'une partie de cet ensemble, ce dernier est validé par l'expérience.

## 3.8 Les architectures de test

Les standards de protocoles de l'*ISO* définissent le comportement permis d'une entité d'un protocole en termes d'*UDP* (Unité de donnée du protocole) et des Primitives de Service Abstraites (*PSA*) aux frontières du service supérieure et inférieure. La méthodologie de test de conformité de l'*ISO* emploie le principe de contrôle et d'observation des primitives de service des couches et des *UDP* à des points spécifiques. Ce principe résulte en quatre méthodes de test de base et leurs variantes. Chacune des méthodes peut être décrite en termes de deux fonctions de test abstraites, le *testeur inférieur* et le *testeur supérieur*, qui peuvent éventuellement être reliées par une forme de *procédures de coordination des tests*. Dans chacun des cas, l'*IST* réside en haut de toutes les entités du protocole testées dans une ou plusieurs couches inférieures, appelées *fournisseurs de service*. Le testeur inférieur exerce son contrôle et son observation des événements de test à la frontière inférieure de l'*IST* à travers les *points de contrôle et d'observation* (*PCO*).

Le rôle du testeur inférieur est d'interpréter les cas de test abstraits et de formuler des verdicts selon les comportements spécifiés dans le cas de test. Ce dernier agit comme une entité paire de l'*IST* et échange des *N-UDP* avec l'*IST.*

Les architectures de test décrivent comment l'*IST* doit être testée, i.e., quelles sorties de l'*IST* sont observées et quelles entrées de l'*IST* peuvent être contrôlées. Les différentes architectures de tests diffèrent selon la nature du testeur supérieur et des procédures de coordination de test associées.

Ces architectures utilisent un ou deux *PCO*s. Dans le cas de deux *PCO*, un est en dessous de l'*IST* est l'autre est au dessus. Sinon, le *PCO* inférieur est utilisé pour contrôler et observer l'*IST.* Même si le *PCO* supérieur est disponible, ceci n'implique pas toujours que le contrôle et l'observation à cette interface de l'*IST* est réalisé à partir du système sous test *(SST)*.

Architecture locale

Dans cette architecture, le testeur supérieur et les procédures de coordination des tests se trouvent dans le système de test. Dans toutes les autres architectures, le testeur supérieur se trouve dans le *SST*. Le *SST* est le système dans lequel l'*IST* doit être exécutée.

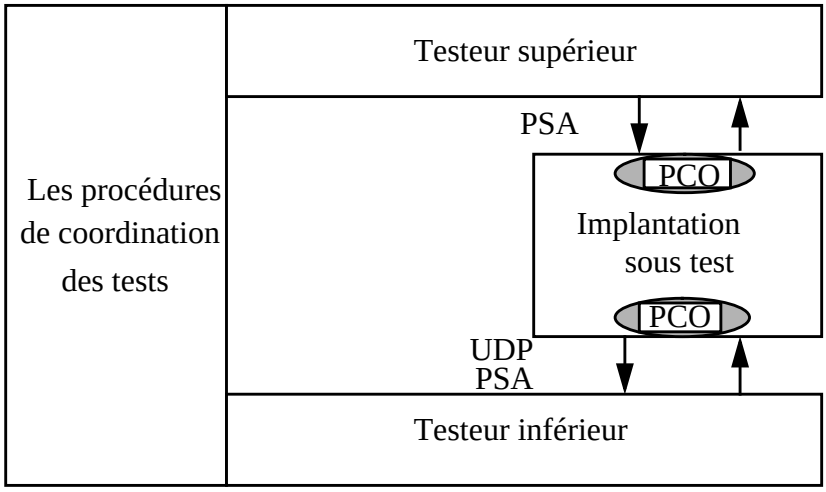



Figure 3.3 Architecture de test locale

Architecture distribuée

Dans ce type d'architecture, l'interface à la frontière supérieure de l'*IST* peut être une interface usager humaine ou bien une interface écrite dans un langage de programmation standard. Dans cette méthode, ainsi que dans la méthode locale, les besoins relatifs aux procédures de coordination des tests sont spécifiés, mais pas leur méthode de réalisation.

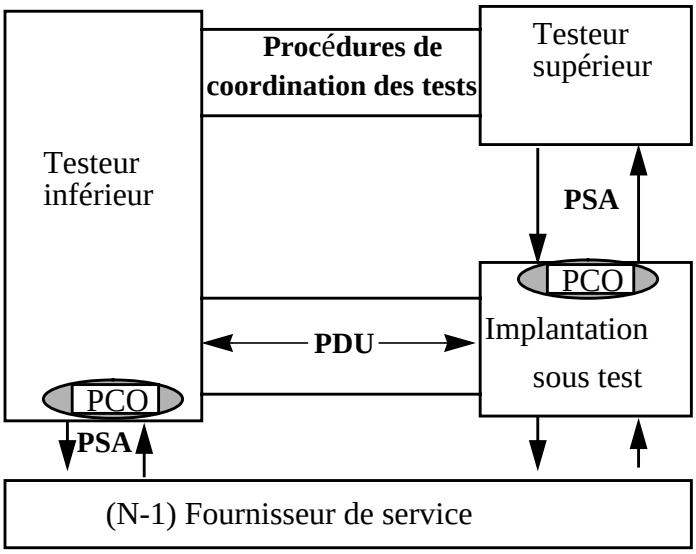



Figure 3.4 Architecture distribuée

Architecture coordonnée

Les procédures de coordination des tests sont normalisées sous forme de *protocole de gestion des tests* (*PGT*). Même si un testeur supérieur se trouve au dessus de l'*IST*, aucun *PCO* n'est utilisé dans cette méthode. Le testeur supérieur est contrôlé et surveillé par le *PGT*, enlevant ainsi le besoin d'accès extérieur à la frontière supérieure.

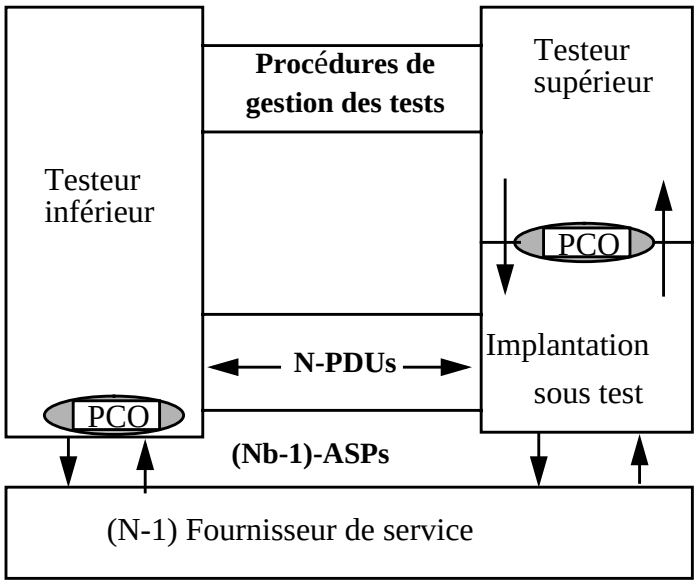



Figure 3.5 Architecture coordonnée

Architecture distante

Les procédures de coordination des tests sont exprimées d'une manière informelle, afin que seuls leurs effets désirables soient décrits dans la suite de test abstraite correspondante. Seul le *PCO* au dessous du testeur inférieur est disponible. Aucun testeur supérieur n'est utilisé par cette méthode. Cependant, pour la spécification des cas de test, certaines fonctions du testeur supérieur peuvent être présentes et utilisées si nécessaire. Les fonctions du testeur supérieur sont réalisées par le *SST* en utilisant tous les moyens possibles pour aboutir aux résultats escomptés.

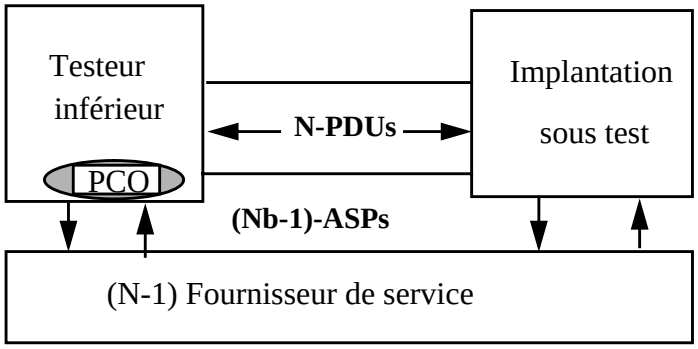



Figure 3.6 Architecture distante

## 3.9 La construction des tests des protocoles de communication

En général, tous les aspects du logiciel classique restent valables pour le test des protocoles de communication. Cependant, ce dernier possède des caractéristiques propres comme par exemple l'aspect interactif inhérent à la communication avec des contraintes temporelles sous-jacentes (synchronisation) et le fait que le protocole nécessite une entité de protocole homologue pour pouvoir fonctionner.

Pour tester le bon fonctionnement d'une implantation d'un protocole de communication, on a besoin d'y appliquer un ensemble de séquences d'entrées appelé *suite de tests* et de vérifier l'exactitude des sorties associées et leur synchronisation. Chaque séquence d'entrée est appelée *cas de test;* elle sert à vérifier une ou plusieurs fonctions particulières du protocole. Un cas de test est généralement composé de trois principales parties (sous-séquences) appelées le *préambule*, le corps du test et le *postambule*. Le préambule est une séquence d'entrée permettant de positionner le protocole dans un état permettant de tester la fonctionnalité visée par le cas de test. Le corps du test permet de tester la fonctionnalité en question. Le postambule vise à réinitialiser le système en vue d'appliquer les prochains cas de test. Il existe plusieurs méthodes de génération des suites de test. Ces méthodes dépendent fortement du formalisme utilisé pour la production de la spécification détaillée [Gone 70, Nait 81, Chow 78, Sabn 88, etc].

Les plus importantes classes de tests sont les tests fonctionnels (boîte noire) et les tests structurels (boîte blanche). La seule façon à ce jour de prouver qu'un système est exact est de le tester exhaustivement (fonction et structure), en fonction du modèle de fautes choisi. Or, les tests fonctionnels exhaustifs sont impraticables car ils impliquent le traitement d'un très grand nombre de cas. Le même problème se pose pour les tests de structure car la vérification de tous les chemins d'un programme entraîne une explosion combinatoire. Étant donné ces problèmes, certaines recherches se sont intéressées à trouver des méthodes de tests pour la validation des aspects les plus importants du logiciel et pour la détection d'un maximum d'erreurs, au lieu de chercher à affirmer l'exactitude du système. Pour cela, il faut se fixer des critères de test à atteindre comme par exemple le taux de fautes couvertes, le taux de chemins vérifiés, etc. De cette façon, l'efficacité d'une méthode de test peut être déterminée par des mesures de couverture appropriées qui évaluent les résultats de l'application des tests par rapport aux objectifs fixés. Il existe plusieurs façons d'appréhender la couverture de test, chacune d'elles apporte un éclairage différent sur le système. Deux classes regroupent ces approches: les méthodes fondées sur la structure de la spécification et les méthodes basées sur les fautes.

## 3.10 Couverture des tests

Dans [Char 96], les auteurs ont choisi de regrouper ces façons en trois catégories: les méthodes fondées sur des éléments de la structure de la spécification, les méthodes fondées sur les fautes et enfin les techniques d'identification.

Les techniques fondées sur des éléments de la structure de la spécification sont issues des méthodes de test des programmes séquentiels [Weyu 85]. Plusieurs auteurs les ont adaptées au cas des protocoles et ont ainsi défini des critères de couverture de spécification propres aux systèmes réactifs, comme par exemple le pourcentage de transitions exécutées ou le nombre d'états couverts ou bien les *io-dataflow-chains* [Ural 91] (seront expliqués plus tard). Les travaux de [Vuon 91] permettent d'exprimer la mesure de la couverture par la qualité de l'exploration de la spécification lors du test. La méthode consiste à munir l'ensemble des traces d'une distance et de prouver que l'espace métrique ainsi obtenu est compact. Ceci veut dire, qu'à partir d'un ensemble de traces on ne prend qu'un

représentant. Ceci permet d'extraire de tout recouvrement de l'ensemble des traces un recouvrement fini. Si l'on considère un recouvrement par des boules de rayon $R$ et que l'on suppose que pour tester les éléments d'une boule il suffit de tester un seul élément de cette dernière alors la couverture peut être exprimée par (1 - $R$). Si $R = 0$, ceci signifie que toute les traces ont été testées et donc la couverture est totale.

La couverture de fautes caractérise l'aptitude d'un test à détecter des fautes dans l'implantation [Boch 91d, Brin 93, Petr 94]. Cette approche est issue du test du matériel où l'on connaît mieux le processus de développement et de fabrication que pour le logiciel. En particulier, il est possible de dresser la liste des types de fautes qui ont pu être commises; c'est ce qu'on appelle le modèle de fautes. Ces techniques ont été adaptées aux systèmes réactifs tel que les protocoles: pour chaque faute recensée dans le modèle, on regarde s'il existe un test qui la détecte. On définit alors la couverture de fautes d'un ensemble de tests comme le nombre de fautes détectées sur le nombre total de fautes. Concrètement, il existe plusieurs façons pour calculer effectivement la couverture de fautes. La plus connue est certainement le calcul par mutation [Howd 82], où on construit réellement une implantation avec chaque faute du modèle et sur laquelle on fait passer les tests. Cette technique bien que relativement fiable est très lourde à mettre en place puisqu'il faut construire chaque mutant et exécuter les tests [Dubu 92]. Les travaux de [Boch 94b, Yao 94] proposent une méthode par dénombrement pour s'affranchir des inconvénients précédents. Des travaux existant sur l'identification des systèmes séquentiels ont conduit [ElMa 93, Zhu 94] à proposer une notion de couverture basée sur cette technique. Le principe de la méthode repose sur le fait qu'un bon test doit être discriminant. En cherchant à identifier toutes les machines qui peuvent passer ces tests, on obtient justement la liste des implantations qui sont indiscernables par le test. On évalue alors la couverture au nombre d'implantations erronées qui ont été identifiées à partir des tests, puisqu'elles contiennent précisément les fautes qui ne sont pas couvertes par le test.

## 3.11 Conclusion

Dans ce chapitre, nous avons fait un survol des différentes stratégies de test et des différents types de test. Nous avons aussi expliqué la différence qui existe entre

vérification et validation. Par la suite, nous avons montré que le test ne peut pas garantir l'absence de fautes, néanmoins, il peut le faire si un modèle de fautes est pris en considération et dans ce cas, le test prouvera la présence ou l'absence de fautes par rapport à ce modèle. Ceci permettra aussi de définir la couverture des tests. Mais avant de pouvoir tester un système, ce dernier doit d'abord être spécifié. La spécification joue un rôle très important dans le cycle de développement d'un produit. Dans le chapitre suivant, nous présentons les méthodes de spécifications les plus utilisées ainsi que les techniques de description (ou spécification) formelles. L'écriture de la spécification permet une compréhension approfondie du système à développer. Elle met en évidence la plupart des ambiguïtés laissées dans l'ombre par les spécifications informelles. Une fois l'investissement d'écriture d'une spécification formelle effectué, nous disposons d'un texte exploitable par des outils de preuve ou d'évaluation symbolique. Nous pouvons alors valider la spécification en prouvant des propriétés souhaitées, ou en réfutant des propriétés correspondant à des situations interdites. Par ailleurs, nous pouvons aussi dériver une maquette, de la spécification formelle, qui en réalise partiellement les fonctionnalités spécifiées, et tester celles-ci très tôt dans le développement. La spécification formelle peut aussi être raffinée. Cette activité s'appelle parfois la réification ou l'implantation abstraite.

Après avoir présenté le test de logiciel, nous passons en revue dans le prochain chapitre les méthodes de génération de tests existantes se basant sur le modèle de *FSM* et *EFSM*, ainsi que certains outils de génération de cas de test se basant principalement sur des systèmes décrits à l'aide du langage *SDL*. La plupart de ces outils sont semi-automatiques et nécessitent que les buts de test soient décrits par l'usager.

# Chapitre 4

# Les méthodes de génération de tests: état de l'art

## 4.1 Introduction

Les algorithmes de génération de cas de test pour les implantations des protocoles constituent une des grandes parties de littérature dans le domaine des logiciels de communication. La tâche de concevoir de tels algorithmes devient plus facile quand les *TDFs* sont utilisées.

Plusieurs méthodes de génération de cas de test fondées sur les spécifications sous forme de *FSM* ont été développées. En y mettant quelques limitations, les résultats de ces méthodes sont directement applicables aux *EFSM*s et aux *TDFs*. Cependant, ces limitations peuvent constituer des restrictions pour la spécification des services, qui, à la demande des usagers, ne cessent de devenir plus complexes. Ce sujet constitue un problème de recherche ouvert qui doit être traité dans le contexte des techniques algorithmiques de génération de tests, même si ce problème est souvent rencontré et résolu par les concepteurs des systèmes de test et des suites de test.

## 4.2 Problème du test des FSMs

Avant de présenter les techniques de génération de cas de test pour les *FSM*s, regardons d'abord pourquoi nous testons les transitions des *FSM*s.

Soit T une transition d'une *FSM* de l'état $s_i$ à l'état $s_j$, causée par la donnée d'entrée I1 et qui génère la donnée de sortie O1.

Soit *IST* une implantation sous test de la *FSM*; il existe trois étapes pour tester si la transition T est implantée correctement:

- Etape 1: Ramener l'*IST* à l'état $s_i$
- Etape 2: Appliquer la donnée d'entrée I1 et observer que l'*IST* génère la donnée de sortie O1
- Etape 3: Vérifier que l'état final de l'*IST* est $s_j$

En général, ces étapes ne sont pas faciles à réaliser du fait de la contrôlabilité et de l'observabilité limitées de l'*IST.*

Pendant l'étape 1, plusieurs transitions peuvent être nécessaires afin de ramener l'*IST* de l'état initial à l'état $s_i$. A l'étape 3, il est typiquement impossible de vérifier que l'*IST* est à l'état $s_j$.

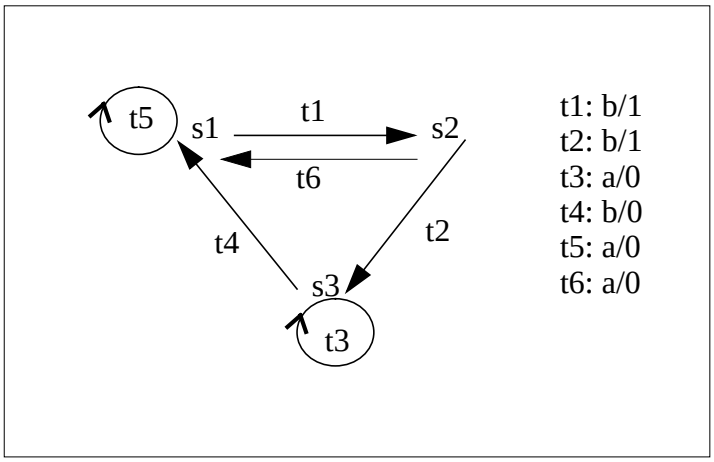



Figure 4.1 Exemple de FSM

Considérons la machine de la figure 4.1. Nous désirons voir les réponses de l'état s1 aux séquences en entrée a. Supposons que la machine se retrouve à l'état s1 après une séquence de transfert (i.e., nous avons appliqué une sequence de messages en entrée jusqu'à ce que la machine se retrouve à l'état s1) et qu'on applique la donnée en entrée a et qu'on observe la donnée de sortie 0. A cause de la présence de fautes, il n y a pas de garanties que la machine se trouvait à l'état s1 en premier lieu (après la séquence de

transfert). Donc il est possible que la machine était à l'état s3.

Toutes les méthodes existantes essaient de présenter des solutions aux problèmes de l'observabilité et de la contrôlabilité. Dans la suite, nous allons décrire quelques une des méthodes (les plus populaires) et nous allons voir comment chacune d'elles résout ces problèmes.

## 4.3 Le modèle de fautes basé sur les FSMs

Dans le modèle *FSM*, nous supposons que la machine est complètement spécifiée, i.e., que les fonctions de sortie et du prochain état sont définies pour tous les états et pour toutes les valeurs des paramètres en entrée.

Les fautes considérées dans ce modèle sont les suivantes:

- Faute de sortie: une transition a une faute de sortie si pour l'état correspondant et la donnée d'entrée reçue, la machine produit une sortie différente de celle qui est spécifiée par la fonction de sortie.

- Faute de transfert: une transition possède une faute de transfert si pour l'état correspondant et la donnée d'entrée reçue, la machine entre dans un état différent de celui spécifié par la fonction de transfert.

- Faute de transfert avec états additionnels: dans la plupart des cas, on suppose que le nombre d'états du système n'est pas augmenté dû à la présence de fautes. Certains types de fautes peuvent seulement être modélisés par des états additionnels, avec des fautes de transfert, ce qui cause la création de ces états additionnels.

- Transitions manquantes ou additionnelles: dans plusieurs cas, on suppose que la *FSM* est déterministe et complètement spécifiée, i.e., pour chaque paire d'état courant et de donnée d'entrée, il y a exactement une seule transition spécifiée. Dans le cas de machine non complètement spécifiée, aucune transition n'est spécifiée pour une certaine

paire, alors que dans le cas de machine non déterministe, plusieurs transitions peuvent être définies. Dans ces cas, le modèle de fautes peut inclure des transitions additionnelles et/ou manquantes.

## 4.4 Hypothèses sur le test des FSMs

Pour le test des FSMs, les hypothèses à faire peuvent être classées en deux classes [Yao 95]: la première classe concerne les propriétés désirables de la spécification alors que la seconde s'intéresse aux types de fautes (e.g., le modèle de fautes [Boch 92, More 90]) qui peuvent être présentes dans l'implantation. Sans la deuxième classe d'hypothèses, toute FSM peut être considérée comme une implantation d'une certaine spécification et l'univers des implantations possibles sera infini. C'est pour cette raison que les hypothèses de la deuxième classe sont introduites pour limiter le nombre d'implantations à considérer [Gill 62, Koha 78].

Concernant les propriétés structurelles, la spécification doit être:

- déterministe
- complètement spécifiée
- fortement connectée ou initialement connectée
- réduite

Pour ce qui est de l'implantation, cette dernière doit:

- être déterministe
- être complètement spécifiée
- avoir un nombre limité d'états additionnels
- avoir un "reset" fiable (un "reset" est un message qui ramène l'*IST* à l'état initial).

## 4.5 Les méthodes de dérivation de cas de test pour les FSMs

Malgré l'utilisation de techniques de descriptions formelles pour la spécification d'un système, il est possible que deux implantations issues de la même spécification ne soient

pas compatibles. Ceci peut avoir plusieurs causes: une mauvaise interprétation de la spécification ou bien des fautes introduites lors de l'implantation. Il est donc nécessaire de tester la conformité de chaque implantation à sa spécification. Le test est réalisé en utilisant des séquences de test.

Pour le test, deux aspects sont à considérer: l'aspect contrôle et l'aspect données. Il s'avère donc nécessaire de tester ces deux aspects surtout pour tester les nouvelles applications et les logiciels de communication. La plupart des méthodes de génération de cas de test traitent soit le flux de contrôle soit le flux de données des spécifications formelles des protocoles et ce d'une manière exclusive.

Dans les années passées, plusieurs méthodes de génération de cas de test ont été proposées pour le test du flux de contrôle. Le rôle du test de flux de contrôle est d'assurer que l'*IST* se comporte comme spécifié par la *FSM* représentant le système. L'objectif est de trouver entre autres les erreurs d'opération ou de transition (erreurs dans la fonction de sortie) et les erreurs de transfert (erreurs dans la fonction ProchainEtat) dans l'*IST*. L'avantage de telles méthodes est la possibilité de les automatiser.

Les méthodes les plus populaires pour la génération de tests pour les *FSM*s sont les suivantes:

Le tour de transitions [Nait 81]

> C'est l'approche la plus directe pour la génération des tests de conformité. Cette méthode consiste à chercher dans la spécification une séquence de transitions appelée tour de transitions. La particularité d'une telle séquence est qu'elle passe par chaque transition de la spécification au moins une fois. Cette méthode nécessite que la *FSM* soit fortement connectée et complètement spécifiée. Elle a cependant une couverture de fautes limitée car elle ne peut garantir la détection des fautes de transfert. Cette technique ne permet pas l'identification de l'état d'arrivée d'une transition. La couverture de fautes des tests générés par le tour de transitions est pire que celle obtenue avec les autres méthodes qui seront présentées.

Dans l'exemple de la figure 4.1, un tour de transitions possible est formé de la séquence de transitions t1, t2, t3, t4, t5, t1, t6 (séquence en entrée: bbababa et séquence en sortie: 1100010). Nous pouvons clairement remarquer que si nous appliquons la séquence en entrée **ba** et si nous observons la séquence en sortie **10**, nous ne pouvons pas savoir quel est l'état courant de l'*IST*. Est-ce que c'est l'état s2 (après avoir exécuté t1 puis t6) ou bien s3 (après avoir exécuté t2 puis t3)?

L'idée de base du test de *FSM* est de vérifier si la transition génère la sortie attendue et si **l'état final est correctement atteint** après la transition. Cependant, à cause de l'observabilité réduite de l'*IST*, l'état atteint est souvent vérifié par une séquence de caractérisation. Dans ce qui suit, les méthodes existantes de génération de tests basées sur certaines méthodes d'identification d'états sont présentées. Ces dernières permettent de répondre à des questions comme celle posée plus haut.

Méthode des séquences uniques d'entrées-sorties (*UIO*) [Sabn 88]

Cette méthode consiste à déterminer pour chaque état de la spécification une séquence d'entrée qui permet de le distinguer de tous les autres états. Ces séquences sont appelés les *UIO* (*Unique Input-Output*). Cette méthode permet de distinguer les états de la *FSM* suivant leur réaction aux *UIOs*. La méthode des *UIO* nécessite que la *FSM* soit fortement connectée, complètement spécifiée et possédant des *UIOs*. La méthode *UIO* offre une couverture de fautes complète, i.e., il n'y a pas d'implantations fautives avec au plus le même nombre d'états que la spécification qui peuvent réussir un test généré par cette méthode.

Dans la figure 4.2 et la table 3 suivantes, nous présentons un exemple de *FSM* et des séquences *UIO* pour chacun de ses états respectivement.

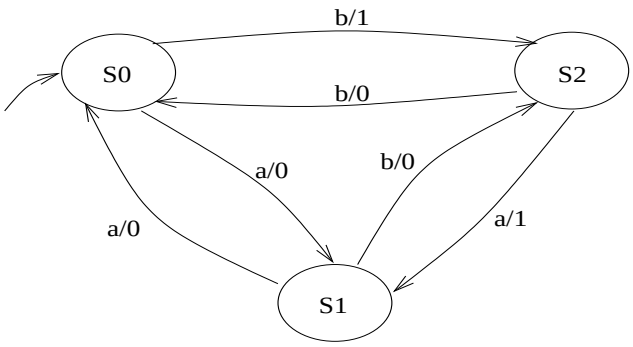



Figure 4.2 Exemple de FSM

| État Si | UIO(Si) | Préambule (Si) |
|---|---|---|
| S0 | b / 1 | Nul |
| S1 | a / 0. b / 1 | a / 0 |
| S2 | a / 1 | b / 1 |

**TABLE 3. Sequence UIO pour la FSM de la figure 5.2**

Méthode *UIO*v [Vuon 89]

Cette méthode est une variante de la méthodes des *UIO*. En général identique à l'*UIO*, mais ajoute une étape de vérification. Cette vérification permet de vérifier si les *UIO* extraites de la spécification sont valables pour l'implantation.

Méthode *W* [Chow 78]

Cette méthode consiste à construire deux ensembles *W* et *P* où *W* est l'ensemble de séquences d'entrée permettant de distinguer entre les états de la *FSM* et *P* l'ensemble de couverture des transitions de la *FSM*, c'est-à-dire l'ensemble des séquences d'entrée qui permettent de passer de l'état initial de la *FSM* à l'état de départ de la transition à tester. Une certaine concaténation des deux ensembles forme l'ensemble des séquences de test. Cette méthode permet de résoudre le problème des *FSM*s qui n'ont ni des séquences de distinction (sera vue plus loin) ni

des séquences *UIO*. Elle exige par contre que la spécification sous forme de *FSM* soit réduite, complètement spécifiée, déterministe et fortement connectée.

Nous présentons un exemple de *FSM* (voir figure 4.3) ainsi que les cas de test correspondants générés par la méthode *W* (voir table 4).

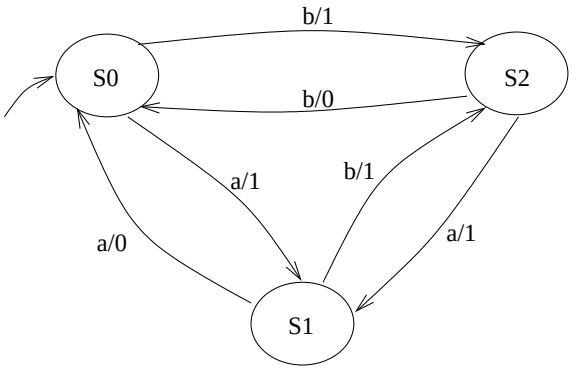



Figure 4.3 Exemple de FSM

| **État Si** | **Sortie pour W = {a,b}** | **Préambule(Si)** |
|---|---|---|
| S0 | {1,1} | Nul |
| S1 | {0,1} | a/1 |
| S2 | {1,0} | b/1 |

**TABLE 4. Un ensemble W pour la FSM de la figure 5.3**

Méthode *Wp* [Fuji 90]

Cette méthode est une optimisation de la méthode *W*. Elle permet d'obtenir des séquences de test de longueur inférieure ou égale à celle de la méthode *W*. Au lieu d'utiliser l'ensemble *W* pour tester chaque état atteint, elle se limite à un sous-ensemble de *W* qui dépend de l'état atteint. Le sous-ensemble ainsi utilisé ($W_i$) s'appelle *ensemble d'identification* de l'état.

Cette méthode consiste en deux phases:

- La première phase vérifie que tous les états définis par la spécification peuvent être identifés dans l'implantation. De plus, les transitions menant à ces états sont aussi vérifiées (voir si leur sortie est correcte ou pas).

- La deuxième phase vérifie que les transitions restantes sont correctement implantées.

Considérons à nouveau l'exemple de la figure 4.3. L'application de la méthode *Wp* aboutit aux ensembles P, Q et $W_i$ (i = 0, 1, 2) ainsi que leurs sorties correspondantes tels qu'illustré dans la table 5. Au lieu d'utiliser l'ensemble W = {a, b} pour vérifier les états S1 et S2, nous utilisons les sous-ensembles W1 = {a} et W2 = {b} pour vérifier ces derniers. Cependant, nous utilisons tout l'ensemble W = {a, b} pour vérifier l'état atteint S0.

| **État Si** | **Sortie pour W = {a, b}** | **Préambule(Si)** | **Ensemble Wi** | **Sortie pour Wi** |
|---|---|---|---|---|
| S0 | {1, 1} | Nul | {a, b} | {1, 1} |
| S1 | {0, 1} | a / 1 | {a} | {0} |
| S2 | {1, 0} | b / 1 | {b} | {0} |

**TABLE 5. Les ensembles W et Wi pour la FSM de la figure 5.3**

Séquence de distinction: SD[Gone 70]

Cette méthode consiste à chercher dans la spécification une séquence d'entrée appelée séquence de distinction. Cette séquence permet de distinguer tous les états de la *FSM*. Selon l'état à partir duquel elle est appliquée, elle donne une séquence de sortie distincte. Cette méthode nécessite que la *FSM* soit fortement connectée, complètement spécifiée et possédant une séquence de distinction.

La SD est une séquence de test qui consiste en trois parties:

- séquence initiale qui ramène la machine à un état spécifique,

- séquence de reconnaissance d'état qui affiche la réponse de chaque état à la *SD*,

- séquence de vérification de transition qui vérifie toutes les transitions de la machine.

Les usagers de cette méthode doivent s'assurer que la *FSM* possède une *SD*. La *SD* est similaire à l'ensemble *W* mentionné ci-dessus. C'est une séquence d'entrée, pour laquelle, pour chaque état initial, la machine produira une séquence de sortie différente.

La figure 4.4 et la table 6 suivantes montrent une *FSM* possédant une *SD* = a.a ainsi que les cas de test générés par la méthode *SD* respectivement.

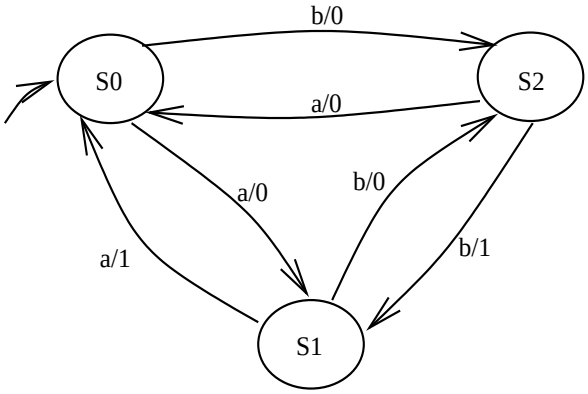



Figure 4.4 Exemple de spécification FSM

| État Si | Sortie pour SD = a.a | Préambule(Si) |
|---|---|---|
| S0 | 0.1 | Nul |
| S1 | 1.0 | a / 0 |
| S2 | 0.0 | b / 0 |

**TABLE 6. Une SD pour la FSM de la figure 5.3**

La figure 4.5 montre par contre une *FSM* ne possédant pas de *SD*.

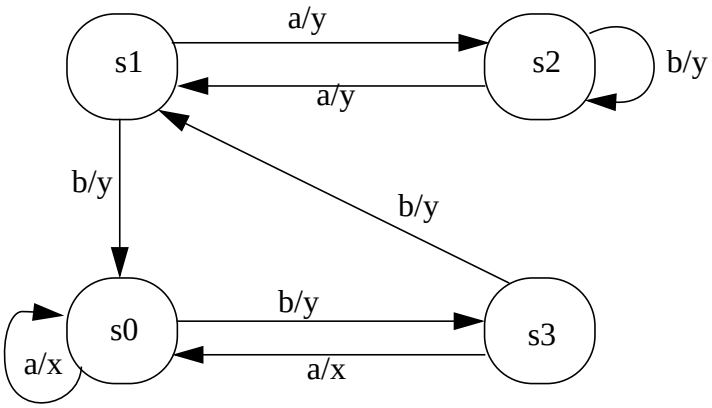



Figure 4.5 Exemple de FSM ne possédant pas de SD (s0 est l'état initial)

Méthode *PSC* [Vuon 90]

Cette méthode offre une vue différente de la génération de séquences de tests en appliquant une technique basée sur les problèmes de satisfaction de contraintes (*PSC*s) utilisés en intelligence artificielle. Dans cette approche, chaque sous-séquence additionnelle est générée en considérant les contraintes imposées sur la structure de la *FSM* par les sous-séquences générées précédemment. Ainsi la séquence de tests est générée d'une façon incrémentale à partir d'une sous-séquence de test initiale de façon à satisfaire l'ensemble de contraintes qui identifie d'une façon unique la *FSM*. Dans les méthodes conventionnelles, les sous-séquences sont générées indépendamment les unes des autres. La séquence de test produite par cette méthode possède une couverture de fautes égale ou supérieure à celle offerte par les méthodes existantes décrites précédemment. De plus, cette méthode offre une procédure naturelle pour la détection d'erreurs lors de l'analyse des résultats de tests et permet de donner une mesure de couverture de fautes pour les séquences de tests.

Dans [Sidh 88], les auteurs comparent, grâce à la technique de Monte Carlo, les méthodes de génération de tests pour les *FSM*s présentées ci-haut. Ils ont conclu que les capacités de détection de fautes des méthodes *UIO*, *SD* et *W* sont les mêmes et qu'elles

sont meilleures que celles du *Tour de transitions*.

Dans ce qui précède, nous avons vu plusieurs méthodes pour le test du flux de contrôle. Cependant, les avantages de la méthode *UIO* sont plus appréciables. Premièrement, le coût d'une séquence *UIO* n'est jamais supérieur à celui de la séquence de distinction et en pratique, il est souvent plus bas. Deuxièmement, presque toutes les *FSM*s possèdent une séquence *UIO* pour chaque état alors que peu ont une séquence de distinction. Aussi, la séquence *UIO* a une avance sur les autres méthodes dans le sens que c'est une approche de génération automatique de cas de test. Cette technique met l'accent sur le test de chaque état plutôt que sur le test de toute la machine. Tout système qui peut être spécifié par une *FSM* peut être testé avec cette technique. Un grand avantage de cette technique est qu'il est possible de construire des séquences de test courtes car même si chaque état à une séquence *UIO* différente, la majorité des séquences *UIO* ont une longueur de 1.

Dans la table 7, nous résumons les méthodes décrites précédemment en indiquant pour chacune d'elles sa couverture de fautes ainsi que les suppositions à faire sur la *FSM* à tester. Une méthode a une couverture de fautes complète si elle permet de détecter toutes les fautes d'opération et de transfert. Dans le cas contraire, elle a une couverture de fautes partielle.

| Méthode | Identification d'état | Couverture de fautes | Suppositions sur la FSM |
|---|---|---|---|
| Tour de transitions | Aucune | Partielle | Fortement connectée<br>Partiellement spécifiée |
| PSC | Aucune | Complète | Fortement connectée<br>Minimale<br>Complètement spécifiée |
| SD | SD | Complète | Fortement connectée<br>Minimale<br>Complètement spécifiée |
| UIO | UIO | Complète | Partiellement spécifiée<br>Minimale |
| W [Chow 1978] | Ensemble W | Complète | Fortement connectée<br>Minimale |
| Wp [Fuji 91] | Ensemble W | Complète | Fortement connectée<br>Minimale<br>Déterministe |

**TABLE 7. Méthodes de génération de cas de test pour le test du flux de contrôle**

Les méthodes présentées plus haut sont des méthodes basées sur les *FSM*s, permettent d'analyser le flux de contrôle des spécifications seulement et laissent la partie données des systèmes non testées. Dans ce qui suit, les méthodes utilisées pour le test du flux de données sont décrites. Mais tout d'abord, nous présentons dans la prochaine section la *normalisation des spécifications* qui permet de transformer une spécification initiale en une spécification équivalente afin que cette dernière puisse être utilisée par les méthodes existantes de génération de cas de test. En effet, les spécifications peuvent être écrites dans différents langages, mais les méthodes existantes de génération de cas de test nécessitent l'extraction d'une forme intermédiaire simplifiée comme la *forme normale* [Sari 86].

## 4.6 La dérivation de la forme normale

La plupart des méthodes de test développées pour *SDL* ou *Estelle* ne génèrent pas directement les cas de test à partir de la spécification *SDL*. Ils transforment la spécification en un modèle mathématique approprié (les *FSM*s [Luo 94a], les *EFSM*s ou les *graphes de*

*flux de données* [Ural 87a]) qu'ils utilisent pour l'extraction des cas de test. Cette transformation n'est pas systématique si la spécification n'est pas écrite dans un style dit normalisé [Ural 87a, Sari 93].

La dérivation de la forme normale ou normalisation est une technique d'analyse statique des spécifications *Estelle* et *SDL* [Sari 93]. Elle permet de transformer une spécification initiale en une deuxième spécification dont chacune des transitions est accessible par un chemin unique. La spécification produite a le même comportement et plus de transitions que la spécification initiale. Les objectifs de la normalisation sont d'identifier tous les chemins menant à une transition, de définir un prédicat (évaluation symbolique) pour chacun de ces chemins et de remplacer toutes les variables incluant les paramètres de sortie par leur valeurs symboliques respectives. Dans le cas d'*Estelle*, la normalisation est achevée en ignorant les constructions qui nécessitent une analyse dynamique. Ces constructions sont: les pointeurs de *Pascal*, *priority*, *attach/detach*, et *connect/disconnect.* On assume aussi que les procédures sont non-récursives. Les transitions sont dépliées en considérant des valeurs de variables héritées à partir d'autres transitions. La normalisation de *SDL* est assez similaire à *Estelle*. Parmi les différences, on peut citer l'élimination de la construction *Save* [Luo 94b] et l'extension des procédures. En conclusion, la normalisation permet de rendre plus claires les transitions de la spécification: état de départ, entrée, prédicat, actions, sorties et état d'arrivée. Ceci correspond exactement au modèle des *EFSMs*.

Comme la procédure de normalisation est systématique, nous supposons que toutes les spécifications *SDL* que nous utilisons sont normalisées. Ceci nous permet de travailler indifféremment sur la spécification *SDL* ou l'*EFSM* associée, qui est son modèle générique.

Après les phases de spécification et d'implantation, le test de conformité doit avoir lieu pour déterminer si la spécification est conforme à l'implantation. La génération de cas de test est une phase très importante du processus de test de conformité. La recherche dans ce domaine a mis l'emphase sur deux problèmes: la génération automatique de cas de test et la sélection automatique de données de tests. Dans les prochaines sections, nous passons en revue diverses techniques qui ont été développées pour résoudre ces problèmes.

## 4.7 Modélisation du flux de données

La méthode du test de flux de données choisit des chemins de tests d'un programme selon les endroits de définitions et d'usages des variables dans le programme. Elle se base sur le graphe de flux de données (*GFD*) (voir figure 4.6) qui est un graphe orienté dont les noeuds représentent les unités fonctionnelles du programme et les arêtes représentent le flux des données. Une unité fonctionnelle peut être une instruction, une transition, une procédure ou même un programme.

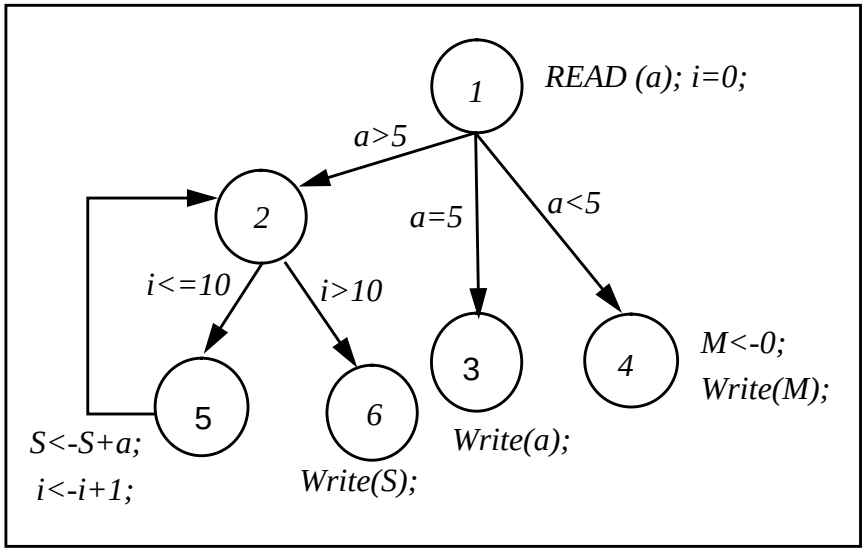



Figure 4.6 Exemple de graphe de flux de données (GFD)

Les méthodes de test du flux de données se basent sur des méthodes qui définissent comment des valeurs sont associées aux variables et comment ces associations peuvent influencer l'exécution d'un programme. Cette analyse s'intéresse aux occurrences des variables dans le programme. Certains auteurs, [Ural 86, Sari 93, Weyu 93, Huan 95, Rama 95] ont proposé de tester quelques propriétés spécifiques associées au flux de données du logiciel. Ces propriétés sont basées sur un ensemble de règles appelées *critères de test.* Chaque critère de test est caractérisé par une couverture particulière des aspects de flux de donnée des protocoles.

Les critères de test du flux de données sont basés sur le choix de chemins qui satisfont certaines caractéristiques de flux de données pour tous les objets de données. Plusieurs critères existent et nous les présentons ci-dessous, brièvement telles qu'ils sont présentés par Rapps et Weyuker [Weyu 85]. Avant cela, présentons tout d'abord quelques définitions de base qui nous seront utiles pour la description des critères de test du flux de données.

**Définitions**

- Une transition possède un **A-Usage** (assignment-use) de la variable x si x apparaît dans la partie gauche d'une instruction de la transition.
- Quand une variable x apparaît dans la liste des paramètres en entrée d'une transition, la transition possède alors un **I-Usage** (input-use) de la variable x.
- Quand une variable x apparaît dans le prédicat d'une transition, on dit que la transition possède un **P-Usage** (predicate-use) de la variable x.
- Une transition possède un **C-Usage** (computational-use) de la variable x si x apparaît dans une primitive de sortie (output) ou bien dans la partie droite d'une instruction d'affectation.
- Une variable x est une définition (ou **def**) si x est un A-Usage ou I-Usage.
- Un **usage global** de x est un usage de x dont la définition apparaît dans une autre transition, autrement, l'usage est local.
- Une **définition globale** de x est une définition de x pour laquelle il y a un usage global dans une autre transition, autrement la définition est locale.
- Un chemin ($t_1$,$t_2$...$t_k$,$t_n$) est un **chemin def-clear** par rapport à la variable x si $t_2$,..,$t_k$ ne contiennent pas des définitions de x.
- Un chemin ($t_1$,...,$t_k$) est un **chemin def-use** (définition-usage) par rapport à la variable x si $x \in def(t_1)$ et soit $x \in C-Usage(t_k)$ ou bien $x \in P-Usage(t_k)$, et ($t_1$,...,$t_k$) est un chemin def-clear par rapport à x de $t_1$ à $t_k$.

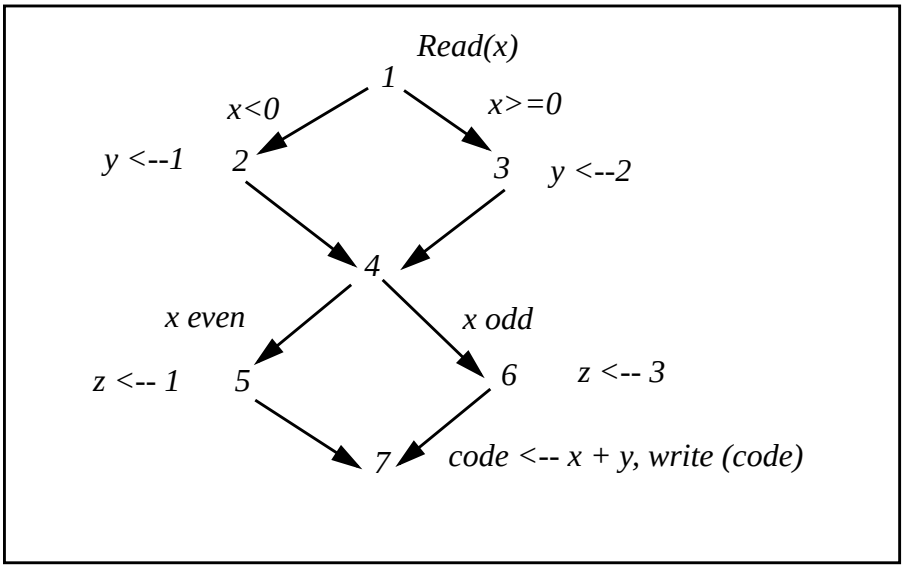



Figure 4.7 Deuxième exemple de graphe de flux de données

Nous définissons maintenant quelques ensembles nécessaires pour la construction des critères de sélection de chemins:

- Def(i) est l'ensemble des variables pour lesquelles le noeud i contient une définition globale.
- C-Usage(i) est l'ensemble des variables pour lesquelles le noeud i contient un usage global.
- P-Usage(i,j) est l'ensemble des variables pour lesquelles l'arête (i,j) contient un p-usage.
- Soit i un noeud quelconque et x une variable tels que $x \in Def(i)$. Alors, dcu(x,i) est l'ensemble de tous les noeuds j tels que $x \in C-Usage(j)$ et pour lesquels il y a un chemin def-clear par rapport à x de i à j.
- dpu(x,i) est l'ensemble de toutes les arêtes (j,k) telles que $x \in P-Usage(j,k)$ et pour lesquelles il y a un chemin def-clear par rapport à x de i à j.

Rapps et Weyuker [Weyu 85] ont proposé une famille de critères pour lesquels le nombre de chemins choisis est toujours fini. Leurs critères s'intéressent plus particulièrement aux types les plus simples de chemins de flux de données qui commencent avec une définition d'une variable et se terminent avec un usage de la même variable. La figure 4.8 présente la famille des critères et leur classification, les noeuds de l'arbre représentent les critères de test et les arcs orientés montrent une relation entre les noeuds fils et les noeuds pères. Cette relation signifie que si une suite de test satisfait le critère père, elle satisfait nécessairement le fils. En d'autres termes les chemins couverts par le critère du noeud fils sont inclus dans ceux du noeud père.

Soit G un graphe de flux de données et P un ensemble de chemins complets de G, alors:

- P satisfait le critère "tous les noeuds" si chaque noeud est inclus dans P.
- P satisfait le critère "toutes les arêtes" si chaque arête de G est incluse dans P.
- P satisfait le critère "toutes les définitions" si pour chaque noeud i et pour chaque $x \in Def(i)$, P inclut un chemin def-clear par rapport à x de i à un élément de dcu(x,i) ou de dpu(x,i).
- P satisfait le critère "tous les p-usages" si pour chaque noeud i et chaque $x \in Def(i)$, P inclut un chemin def-clear par rapport à x de i à tous les éléments de dpu(x,i).
- P satisfait le critère "tous les usages" si pour chaque noeud i et chaque $x \in Def(i)$, P inclut un chemin def-clear par rapport à x de i à tous les éléments de dpu(x,i) et à tous les éléments de dcu(x,i).
- P satisfait le critère "toutes les définition-usages" si pour chaque noeud i et chaque $x \in Def(i)$, P inclut tous les chemins "définition-usage" (ou def-use) par rapport à x. Donc, si plusieurs chemins "def-use" existent d'une définition globale à un certain usage, ils doivent être tous inclus dans P.
- P satisfait le critère "tous les chemins" si P inclut chaque chemin de G.

Il y a toujours un compromis à faire lors du choix d'un critère. Plus le critère est fort, plus la tâche de tester le programme est minutieuse. Cependant, un critère faible peut être satisfait, en général, avec un petit nombre de cas de test.

Toutes les méthodes existantes de génération de tests supposent que les procédures locales ne sont pas récursives et que les boucles ont des bornes constantes.

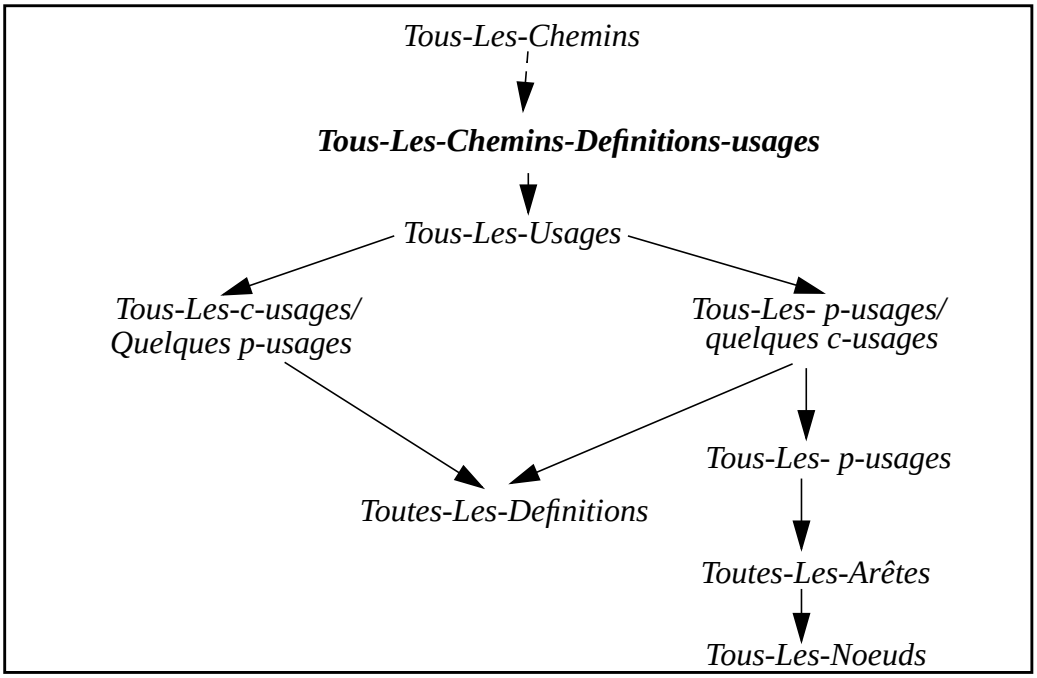



Figure 4.8 Classification des critères de flux de données

## 4.8 Méthodes existantes de génération de tests pour les EFSMs utilisant les techniques de test du flux de données et/ou de test du flux de contrôle

Quand une spécification est sous forme de *EFSM*, les méthodes traditionnelles de test de *FSM* telles que les séquences *UIO*, la méthode *W*, la *DS* ne sont plus suffisantes. La partie "données" doit aussi être testée afin de déterminer le comportement de l'implantation. Certains travaux ont été réalisés dans ce domaine, parmi ceux-ci:

Sarikaya 1987

Cette méthode peut être utilisée pour dériver des tests vérifiant les aspects de flux de données des protocoles de communication. Elle est applicable aux spécifications formelles écrites dans le langage *Estelle*.

La génération de tests est réalisée en quatre étapes. La première étape consiste à transformer la spécification *Estelle* en une spécification sous forme normale. La deuxième étape consiste à construire le graphe de contrôle. Ce dernier est une représentation graphique de la *FSM* sous-jacente. Dans la troisième étape, un graphe de flux de données est construit et les cas de test sont dérivés dans la quatrième étape.

En utilisant des méthodes de décomposition et de partitionnement fonctionnel, les auteurs ont dérivé des blocs *GFD* qui sont considérés comme des fonctions de flux de données (*FFD*). Pour chaque bloc, un tour est choisi à partir du *GFC* (graphe de flux de contrôle). Le tour est choisi tel que ses transitions (en forme normale) couvrent toutes les transitions du bloc fonctionnel. Par la suite, des données de test (valeurs des paramètres des données d'entrée) sont déterminées. Finalement, le tour choisi ainsi que les valeurs pour les paramètres en entrée sont utilisés pour construire une suite de test complète.

Ural 1991

Les auteurs proposent une méthode qui génère des séquences de tests à partir d'une spécification en forme normale (*NFS*: Normal Form Specification) [Sari 87]. D'abord, la *NFS* est transformée en un graphe de flux. Ensuite, les définitions et usages de chaque variable de contexte ainsi que de chaque paramètre en entrée et en sortie sont identifiés. La sélection des séquences de tests est basée sur l'identification et la couverture de chaque association entre un paramètre en sortie et tous les paramètres en entrée qui influencent ce paramètre en sortie (*IO-dataflow-Chains*). La méthode requiert que chaque association soit examinée au moins une fois. Finalement, des séquences de tests sont choisies pour couvrir ces associations

au moins une fois. Cette méthode génère moins de séquences de tests que d'autres méthodes. Elle ne vérifie pas si les séquences de test sont exécutables ou pas. De même, c'est l'usager qui doit construire les séquences à partir des chemins *définition-sortie* générés et la sélection des données de tests n'est pas offerte.

Miller 1992

Dans cette méthode, une version limitée d'*Estelle* est utilisée. Tout d'abord, la *EFSM* est convertie en une *FSM* équivalente avec une modification des entrées et sorties. Le nombre d'états n'est pas augmenté, mais le nombre de transitions l'est. Tout d'abord, un *GFD* est construit à partir de la *FSM*. Ensuite, les chemins qui couvrent le flux de données (critère *toutes les définitions-sorties*) et le flux de contrôle, sont générés et combinés afin de générer des séquences de tests exécutables.

Vuong 1992

Dans cette méthode, les auteurs suggèrent une technique hybride qui combine deux techniques de génération de cas de test pour aboutir à une séquence de test qui couvre séparément le flux de contrôle et le flux de données des protocoles spécifiés avec *Estelle*. Le flux de contrôle est testé en premier, suivi par le flux de données qui utilise une technique de variation de paramètres. La génération et la sélection de cas de test sont guidées par des contraintes sur les suites de test ainsi que sur la variation de paramètres spécifiées par l'utilisateur.

Chanson 1993

[Chan 93] propose une méthode intéressante qui utilise les techniques d'analyse de flux de données. Le critère *toutes les définition-usages* est utilisé pour déterminer toutes les dépendances (de données et de contrôle) entre les transitions. Pour tester le flux de données, il suffit de tester ces dépendances. L'algorithme utilisé teste le flux de contrôle en utilisant la méthode *UIO, W* ou *SD*. Cependant, cette méthode présente des inconvénients:

- Plusieurs séquences de tests générées par cette méthode sont éliminées du fait qu'elles ne contiennent pas la *boucle influente*. Cette situation peut être évitée si l'algorithme de génération des séquences de tests prenait en compte le traitement de ces boucles influentes.

- Après la génération des chemins def-clear, ceux-ci sont fusionnés pour constituer une séquence de tests. Cette fusion peut résulter en des séquences finales non exécutables dû à la non-exécutabilité d'un ou de plusieurs chemins def-clear. Ce problème peut être évité en modifiant l'algorithme présenté dans [Chan 93] en évitant de faire la fusion des chemins def-clear.

Huang 1995

Dans cet article, une méthode de génération de séquences de tests exécutables pour le test du flux de données pour des protocoles modélisés par des *EFSM* est présentée. Pour le flux de données, le critère *toutes les définition-sorties* est utilisé. Une séquence de test exécutable est composée de trois parties: un préambule, un chemin *définition-sortie* et un postambule. Toutes les séquences générées sont exécutables, mais cette technique est une sorte d'analyse d'accessibilité et donc souffre du problème de l'explosion combinatoire.

Ramalingom 1995

[Rama 95] présente une méthode unifiée de génération de cas de test pour les protocoles modélisés par des *EFSMs* en utilisant les séquences uniques indépendantes du contexte (*CIUS*). Cette méthode considère la faisabilité des cas de test pendant leur génération. Un nouveau type de séquence d'identification, Trans-*CIUS*, est défini. Le critère *Trans-CIUS* utilisé dans le test du flux de contrôle permet de générer une séquence unique d'entrée ainsi qu'un préambule exécutable indépendant du contexte pour atteindre l'état de départ, s, de la transition à tester tel que les prédicats de chaque transition du préambule sont indépendants de tout contexte valide à l'état s. Ce critère est supérieur aux critères existants de couverture du flux de contrôle pour les *EFSMs* du fait qu'un *CIUS* d'un certain état

est suffisant pour tester toutes les transitions aboutissant à ce dernier. Afin d'offrir une observabilité, le critère "tous-les-usages" est étendu à ce qui est appelé "def-use-ob" (ob pour observation). Finalement, un algorithme d'exploration en largeur à deux phases est conçu pour la génération d'un ensemble de tours de test exécutables couvrant les critères mentionnés.

Dans la table 8, nous présentons quelques méthodes de génération de tests et les critères (des flux de données et de contrôle) sur lesquels elles se basent. De même, nous mentionnons si les séquences de tests générées par chacune d'elle sont exécutables ou pas.

| **Méthode** | **Critère de couverture pour le flux de données** | **Critère de couverture pour le flux de contrôle** | **Exécutabilité des séquences de tests** |
|---|---|---|---|
| Ural 91 | IO-df-chain | - | N/D |
| Miller 92 | définition-observation | UIO | Exécutables |
| Chanson 93<br>Chanson 94 | définition-usages | DS, UIO, ou W | Quelque séquences de tests exécutables ne sont pas générées |
| Huang 95 | définition-observation | - | Exécutables |
| Ramalingom 95 | définition-observation | CIUS | Exécutables |

**TABLE 8. Méthodes de génération de test pour les *EFSMs***

Après avoir présenté les différentes méthodes de génération de cas de test pour les *EFSMs*, nous présentons dans ce quit suit les méthodes de test pour les systèmes complexes.

## 4.9 Le test des systèmes composés

Jusque là, nous avons vu les méthodes de test pour les systèmes composés d'un seul module (*FSM* ou *EFSM*) et nous avons supposé que le module est testé en isolation. Mais avec le développement des systèmes distribués, il est devenu très important de tester les composantes qui sont *encapsulées* (embeded) dans un système. Tous les composants autres que celui à tester constituent le contexte de test. Dans un tel environnement, le test est réalisé en appliquant les suites de test au niveau système afin de voir le comportement du composant cible puisque les interfaces du module encapsulé ne sont pas directement

accessibles par le testeur.

Le problème du test d'un composant encapsulé (ou bien "gray-box testing") se pose s'il s'agit de tester certains composants du système global. Dans ce cas, le système est représenté comme une composition de deux machines abstraites. La première, appelée composant ou *IST*, est la machine formée de tous les composants à tester. L'autre machine, appelée l'environnement ou le contexte, est la composée de toutes les autres machines que l'on suppose être sans faute. Les deux machines communiquent à travers des interfaces cachées et le contexte communique avec l'environnement à travers les *PCO*s (d'une manière observable). Ce test particulier soulève les problèmes de contrôlabilité et d'observabilité partielles du composant sous test. Dans la plupart des cas et selon le comportement de l'environnement, le produit partiel va offrir moins de possibilités de détection de fautes (moins de contrôle et d'observabilité) que dans le test du composant en isolation.

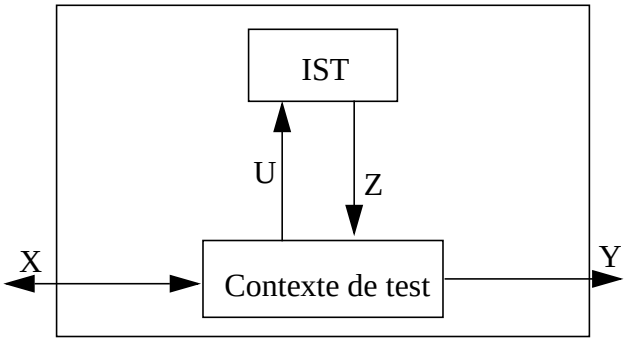



Figure 4.9 Système encapsulé

La plupart des travaux sur le test des protocoles de communication concernent seulement le test d'un seul composant du système où chaque composant est testé en isolation. Récemment, des travaux sur le test dans le contexte sont apparus [Petr 96, Petr 97, Lima 97]. Ces derniers concernent surtout la définition de modèles de fautes et la génération effective de suites de test par rapport aux modèles de fautes donnés. Dans [Petr 96], les auteurs proposent un cadre de travail pour le test dans le contexte. Cette méthode calcule une approximation de la spécification dans le contexte et l'idée de base consiste à

réduire le test dans le contexte au test en isolation. La méthode essaie de donner la solution la plus générale à ce problème. Aussi, dans [Lima 97], les auteurs présentent une approche pragmatique pour la génération de séquences de test pour les composants encapsulés d'un système complexe. L'approche proposée est basée sur:

- La définition d'une procédure de composition qui permet l'abstraction des signaux internes échangés entre les processus composant le système, tout en préservant les échanges entre le système et son environnement. L'algorithme de composition défini permet l'identification des parties du système global qui reflètent le comportement du composant.

- Le test orienté but. Les transitions qui reflètent le comportement du composant peuvent être utilisées pour construire des objectifs de test qui ne testent que l'implantation du composant.

Ces travaux présentés sur le test dans le contexte concernent seulement les systèmes modélisés par des *FSM*s et la partie données de ces systèmes n'est donc pas traitée.

## 4.10 La génération de tests basée sur les *TDF*s

Les *TDF*s sont définies afin de:

- minimiser les ambiguïtés dans les spécifications des protocoles

- automatiser le processus d'implantation et donc automatiser la génération de tests pour ces spécifications.

Typiquement, la tâche de définir des spécifications formelles, implantées d'une manière efficace, requiert qu'une spécification consiste en un petit nombre de modules indépendants, tel que chacun puisse être facile à implanter. Cependant, si la spécification n'est pas conçue d'une manière attentive, une fois que ses modules sont placés dans l'implantation, l'interaction peut rendre le comportement de toute l'implantation non contrôlable et non observable. Un autre problème, concernant les spécifications écrites sous forme de plusieurs processus communicants, est d'obtenir le comportement combiné

de ces processus. Le processus global peut être extrêmement large et le problème de vérification de l'exactitude de la spécification peut devenir impossible même pour des spécifications avec peu de processus.

Dans ce qui suit, nous présentons quelques travaux qui adressent le problème de la génération de tests basée sur les *TDF*s.

- Dans [Brom 89] les auteurs présentent une heuristique pour dériver le comportement global d'un protocole sous forme d'un arbre appelé l'*arbre ACT* (Asynchronous Communication Tree). Ce dernier est basé sur un sous-ensemble de *SDL*. L'*ACT* est la description du système global telle qu'obtenue par l'analyse d'accessibilité avec perturbation. Dans l'*ACT*, les noeuds représentent les états globaux. Un état global contient l'information sur les états de tous les processus de la spécification. Les tests sont dérivés à partir de l'*ACT* d'une spécification par un logiciel appelé *TESDL*.

- Une théorie pour définir le test de conformité pour des systèmes *LOTOS* (ou bien en général pour des systèmes à transitions étiquetées [Miln 80, Hoar 80]) est présentée dans [Brin 87]. Les auteurs définissent la notion de *testeur canonique* qui détecte les inconsistances entre une spécification et son implantation. Cet article représente une méthodologie formelle pour la définition de concepts fondamentaux dans le domaine du test de conformité.

- Une approche plus pratique pour les spécifications *LOTOS* peut être trouvée dans [Guer 90]. Les auteurs appliquent les séquences *UIO* à des protocoles réels écrits dans le langage *LOTOS* où une approche orientée états est utilisée. Les auteurs dérivent des *arbres d'exécution* de la spécification *LOTOS* en enlevant les opérateurs *LOTOS* tels que la composition parallèle et l'opérateur *Disable*. Les arbres d'exécution représentent toutes les séquences d'exécution possibles, dont quelques unes peuvent être invalides. Les tests de conformité résultants, pour le protocole *LAP-B* (protocole de la couche lien de données) sont compatibles avec ceux générés par les tables de transition d'états. Même si les auteurs mentionnent que l'algorithme de sélection des tests est basé sur

une heuristique et peut ne pas être faisable pour des spécifications plus complexes et plus détaillées, cet article est l'un des exemples encourageants qui montrent la combinaison de spécifications formelles avec les techniques de génération des tests.

Comme mentionné plutôt, la sélection des données de test est un des problèmes à résoudre afin de pouvoir générer des cas de test pour les *EFSMs*. Dans la prochaine section, nous présentons les différentes méthodes pour résoudre ce problème.

## 4.11 La sélection des données de test

Le nombre de travaux qui ont été réalisés pour la sélection des données de tests est de beaucoup inférieur à celui de ceux pour la génération des cas de test. Les cas de test générés pour les *EFSM*s ne sont pas nécessairement tous exécutables. Ceci est dû à la présence de prédicats, i.e, conditions ou contraintes, dans les séquences à tester, qui ne peuvent être satisfaits par aucune donnée en entrée.

La sélection des données de test est une étape très importante. Le développement des ensembles de données de test revient à sélectionner les données en entrée et à déterminer les sorties attendues.

A notre connaissance, les méthodes utilisées pour la sélection des données de tests sont les techniques *CLP* (Constraint Logic Programming) et l'analyse de mutation. Avant de présenter ces deux méthodes, nous allons tout d'abord définir l'exécution symbolique qui est utilisée pour interpréter les séquences de tests et pour générer les contraintes à résoudre.

### 4.11.1 L'exécution Symbolique

L'exécution symbolique [Clar 85, Howd 78, King 76] est une méthode d'analyse de programmes qui représente les calculs d'un programme ainsi que son domaine par une expression symbolique. Elle décrit la relation entre les données en entrée et les valeurs résultats, alors que l'exécution normale calcule des valeurs numériques mais ne se rappelle pas de la manière avec laquelle ces valeurs numériques ont été dérivées.

Les variables d'un programme peuvent être divisées en trois ensembles:

- les variables d'entrée sont celles qui sont définies par les données des cas de test,
- les variables de sortie sont celles qui sont accessibles à la suite de l'exécution des cas de test,
- les variables de calcul sont celles qui sont utilisées d'une manière strictement interne dans le programme.

Dans le test normal, des valeurs sont affectées aux données d'entrée. Le programme est alors exécuté. Après l'exécution, les variables de sortie sont examinées afin de déterminer l'exactitude. Par contre, dans l'exécution symbolique, on donne des valeurs aux variables d'entrée. Un chemin de contrôle à travers le programme est décrit. Le programme est alors exécuté symboliquement à travers ce chemin. Après l'exécution, les sorties symboliques sont examinées afin de déterminer l'exactitude du programme. Les chemins de contrôle peuvent être donnés explicitement par l'usager ou générés automatiquement par le système de test.

L'exécution symbolique peut être réalisée de deux façons: l'expansion avant et la substitution arrière. L'expansion avant modélise essentiellement les actions quand les transitions sont réellement exécutées. Elle commence avec le noeud initial et travaille en se dirigeant vers le noeud final, alors que la substitution arrière commence avec le noeud final et travaille en se dirigeant vers le noeud de départ [Clar 85, King 76]. L'expansion avant est normalement utilisée pour le calcul des chemins alors que la substitution arrière est utilisée pour créer la condition du chemin.

Une difficulté majeure pour l'exécution symbolique est la manipulation des boucles (ou itérations). Est-ce que les boucles doivent être évaluées une fois, deux fois, cent fois ou aucune fois? Certains exécuteurs symboliques prennent une approche pragmatique. Pour chaque boucle, trois chemins sont construits, chacun contenant: aucune exécution de la boucle, une seule exécution de la boucle et deux exécutions de la boucle.

La figure 4.10 montre un exemple de programme ainsi que les valeurs symboliques de ses variables.

| | | Condition du chemin | a | b | c | d |
|---|---|---|---|---|---|---|
| 1 | Begin | - | - | - | - | |
| 2 | Read a, b, c, d | - | a | b | c | d |
| 3 | a:= a+b | - | a+b | b | c | d |
| 4 | IF a>c | a+b<=c | a+b | b | c | d |
| 5 | THEN d:=d+1 | | | | | |
| 6 | ENDIF | a+b<=c | a+b | b | c | d |
| 7 | IF b=d | a+b<=c AND b<>d | a+b | b | c | d |
| 8 | THEN WRITE("Success", a,d) | | | | | |
| 9 | ELSE WRITE("Fail", a,d) | a+b<=c AND b<>d | a+b | b | c | d |
| 10 | ENDIF | a+b<=c AND b<>d | a+b | b | c | d |
| 11 | END | a+b<=c AND b<>d | a+b | b | c | d |

Figure 4.10 Fragment de programme et valeurs symboliques pour un chemin

L'exécution symbolique est utilisée dans les méthodes de vérification formelle pour la formulation des conditions qui doivent être démontrées. Elle est aussi utilisée pour le débogage et l'optimisation des programmes ainsi que pour le développement des logiciels (les besoins du programme peuvent être exprimés sous forme de représentations symboliques).

### 4.11.2 Le problème de satisfaction de contraintes (PSC)

Un problème *PSC* [Dech 89] est composé d'un ensemble de n variables $X_1$,..., $X_n$, chacune représentée par les valeurs de son domaine $R_1$,..., $R_n$ et d'un ensemble de contraintes. Une contrainte $C_i(X_{i1},...,X_{ij})$ est un sous-ensemble du produit cartésien $R_{i1}x...xR_{ij}$ qui spécifie quelles valeurs des variables sont compatibles entre elles. Une solution est une affectation de valeurs à toutes les variables qui satisfont les contraintes et la tâche est de trouver une ou plusieurs solutions.

Les *PSCs* sont en général NP-complets. Cependant, des chercheurs tels que [Fike 70, Mont 74, Mack 77] ont montré que certains sous-ensembles des *PSCs* peuvent être résolus

d'une manière efficace, en exploitant les connaissances sur le domaine et en manipulant les contraintes d'une manière intelligente. Les *PSCs* peuvent être résolus par les systèmes *CLP* (constraint logic programming). Un *CLP* est un descendant de la programmation logique qui a connu un très grand succès grâce au langage Prolog.

Maintenant, les langages *CLP* permettent aux programmes logiques de s'exécuter d'une manière très efficace en s'intéressant à des domaines de problèmes particuliers. Un programme *CLP* utilise une base de faits, mais peut utiliser les contraintes pour éliminer plusieurs alternatives, ce qui le rend plus efficace.

### 4.11.3 L'analyse de mutation

L'analyse de mutation n'est pas seulement une méthode de génération de données de test, mais c'est aussi une procédure pour mesurer la qualité ou l'adéquation des cas de test [DeMi 91, Mill 92].

En pratique, un testeur interagit avec un système de mutation automatique pour déterminer et améliorer l'adéquation des cas de test. Ceci est réalisé en forçant le testeur à tester des types de fautes spécifiques. Ces fautes sont des changements syntaxiques simples au programme à tester qui produisent des programmes mutants. Le but du testeur est de créer des cas de test qui différencient chaque mutant du programme original en le poussant à produire des sorties différentes. Autrement dit, le testeur essaie de choisir des valeurs pour les données en entrée qui causeront l'échec de chaque mutant. Quand les sorties d'un mutant diffèrent des sorties du programme original pour certaines entrées, le mutant est considéré comme mort. Un cas de test qui tue tous les mutants est adéquat relativement à ces mutants. En général, il est impossible de tuer tous les mutants car certains changements n'ont aucun effet sur le comportement fonctionnel du programme.

Après avoir présenté les techniques de sélection des données de test, nous présentons ci-dessous les méthodes existantes de génération de cas de test ainsi que la méthode de sélection de données de test utilisée par chacune d'elles.

### 4.11.4 Quelque méthodes de sélection des données de tests

Dans [Ural 91], les auteurs ont mentionné le problème de sélection des données de test, mais ne l'ont pas résolu. Dans [Higa 93], seuls les types "Entier" et "Booléen" sont considérés et les opérateurs se réduisent à "+" et "-". Les techniques de programmation linéaire sont utilisées pour résoudre les contraintes. [Chun 90] a utilisé les techniques *CLP* [Jaff 87] pour générer uniquement des séquences de tests exécutables. L'auteur a mentionné que même si le problème est NP-complet, les techniques *CLP* peuvent être utilisées dans le contexte des protocoles de communication du fait de leur simplicité. Dans [Mill 92], après la génération des séquences de tests, les données de tests sont choisies pour chaque séquence en utilisant une technique de mutation faible qui garantit la détection de certains types de fautes dans le flux de données. Dans [DeMi 91], une technique qui génère automatiquement les données de tests est présentée. Cette dernière est basée, elle aussi, sur l'analyse de mutation et permet de créer des données de tests qui approchent l'adéquation relative. [Chan 93] a utilisé l'analyse de boucle pour les boucles influentes et la méthode *PSC* pour résoudre le problème de l'exécutabilité. La sélection des données de test n'est pas traitée. Cependant, dans [Chan 94], un générateur automatique de cas de test pour les protocoles de communication a été présenté. Tout d'abord, les séquences de tests sont générées en utilisant la méthode décrite dans [Chan 93]. Un ensemble de conditions (ou contraintes) pour chaque séquence de tests est généré en utilisant des techniques d'évaluation symbolique. Finalement, en résolvant ces conditions (comme un système de contraintes), les données de test sont automatiquement générées. Une heuristique pour trouver une solution à un système de contraintes est proposée.

La plupart des outils de génération de tests commerciaux n'utilisent pas les méthodes présentées ci-dessus. Ceci est dû au fait que ces méthodes traitent des spécifications simples et qu'elles ne peuvent pas générer des tests pour des systèmes complexes. Dans ce qui suit, nous présentons quelques outils de génération de tests. Presque tous ces derniers sont semi-automatiques. Cependant, certains outils existent qui automatisent des parties du processus de test. Tout d'abord, nous présentons dans la prochaine section, le langage *TTCN* puisque c'est avec ce langage que la plupart des cas de test pour les protocoles de communication sont généralement écrits.

## 4.12 Le langage *TTCN*

Généralement, nous supposons, en rapport avec l'évaluation de la conformité, qu'un cas de test (ou un scénario de test) est écrit en pensant à un but de test particulier. Si les cas de test doivent être la base du test de conformité de *ISO/OSI* (International Organization for Standardization/Open Systems Interconnection), il est clair qu'une notation ou un langage est nécessaire pour spécifier le comportement d'un système de test et de l'entité de protocole à tester. Le langage doit contenir suffisamment de caractéristiques pour décrire les tests pour tous les protocoles de l'*ISO*. C'est ainsi que *ISO* a défini une notation appelée *TTCN* (Tree and Tabular Combined Notation) [ISO 92].

*TTCN* est le langage standard recommandé par *ISO/IEC* 9646 pour l'écriture des tests abstraits et pour la dérivation de suites de tests à partir de spécifications *SDL* par exemple. *TTCN* ressemble quelque peu au langage *Pascal* ou *C* dans le sens qu'il comporte une partie déclarative et une partie procédurale. Les tests sont écrits sous forme de tables dont la structure est très bien définie. La structure d'un test est hiérarchique; une suite de tests contient un nombre de cas de test qui sont à leur tour composés d'étapes de tests et de *défauts* qui représentent les actions à prendre quand un événement inattendu survient. *TTCN* est abstrait dans le sens qu'il est indépendant des systèmes de test. Ceci veut dire qu'une suite de tests pour une application peut être utilisée dans n'importe quel environnement de cette dernière.

Durant les années passées, l'utilisation de *TTCN* a largement augmenté. Ceci est vrai d'autant plus qu'un grand nombre de suites de tests *TTCN* a été émis par différents organismes de normalisation; aussi, *TTCN* a été très utilisé dans l'industrie du fait qu'il est compatible avec tous les types de test fonctionnel pour les systèmes communicants.

## 4.13 Panorama des outils de génération de tests à partir de spécifications *SDL*

Parmi les principales méthodes de génération de tests qui se sont succédées depuis le début des années 80, une grande partie des études porte sur la génération de tests à partir de modèles mathématiques comme les *FSM*s (et leurs variantes) et les systèmes de transitions étiquetées [Tret 92]. Les méthodes de génération de tests à partir de

spécification *SDL* s'appuient pour la plupart sur ces méthodes:

- soit en transformant la spécification *SDL* en *FSM* [Cava 95]
- soit en calculant l'arbre d'accessibilité de la spécification *SDL* [Hogr 88].

Transformer une spécification *SDL* vers un modèle mathématique permet d'appliquer les nombreuses méthodes concernant ces modèles. Cependant, les modèles obtenus sont souvent de très grande taille ce qui génère un nombre important et donc inexploitable de cas de test; Aussi, pour les systèmes réels, la transformation en *FSM*s ou la génération du graphe d'accessibilité sont pratiquement impossibles.

L'avantage des méthodes basées sur le graphe d'accessibilité est de pouvoir générer des tests complets. Pour faire face au problème de l'explosion du graphe, certaines méthodes proposent de construire un sous-ensemble des chemins de l'arbre d'accessibilité répondant aux buts de tests. Ces derniers peuvent être représentés sous forme de *MSC* ou bien dérivés automatiquement de la spécification *SDL*.

Les prochaines sections décrivent les outils existants de génération de cas de test.

### 4.13.1 SAMSTAG [Grab 94]

*SaMsTaG* (Université de Bern) est l'abréviation de "*SDL* And *Msc* baSed Test cAse Génération". Cette méthode permet de générer des cas de test abstraits à partir d'une spécification formelle et d'un ensemble de buts de test. Le comportement du protocole à tester est donné sous forme d'une spécification *SDL* et les buts de test sont donnés par des *MSC*s (Message Sequence Charts) qui sont un moyen graphique très connu pour visualiser certaines exécutions de systèmes de télécommunication et sont décrits manuellement. Cette méthode consiste à réaliser une composition parallèle de la spécification *SDL* et du *MSC* par simulation. *SaMsTaG* consiste en un outil de simulation de *MSC*, d'un outil de simulation de systèmes *SDL* et d'un générateur de cas de test. Cette méthode nécessite que l'usager entre les buts de test.

### 4.13.2 Autolink [Schm 98]

Autolink est un outil de génération automatique de tests. Il permet de générer des suites de test *TTCN* à partir d'une spécification *SDL* et de besoins formulés sous forme de *MSC*. Autolink a été influencé par l'outil *SAMSTAG*. La génération de cas de test passe par plusieurs étapes. Tout d'abord, l'usager doit définir des chemins *SDL* qui serviront de base pour la génération des cas de test. Chaque chemin est enregistré sous forme de *MSC*. Ce dernier peut montrer uniquement le comportement observable du système avec son environnement. Ensuite, il faut définir une configuration car une suite de test dépend de plusieurs options (e.g., l'usager peut choisir parmi plusieurs formats pour les données de sortie). En se basant sur les *MSC* et le fichier de configuration, Autolink calcule une représentation interne pour chaque cas de test. Cette représentation contient toutes les séquences d'envoi et de réception de messages qui entraînent un verdict "pass" ou "inconclusive". De plus, elle garde une trace de la structure du cas de test. La dernière étape consiste à générer les suites de test *TTCN* à partir de la représentation interne des cas de test et de la liste de contraintes.

### 4.13.3 Tveda V3 [Clat 95]

*Tveda V3* est un outil pour la génération automatique de cas de test à partir de spécifications écrites dans les langages *Estelle* ou *SDL*. Cette méthode propose de calculer un ensemble de buts de test sur le graphe syntaxique de la spécification *SDL* (un but de test correspondant à une transition du graphe *SDL*). Les tests générés par cet outil sont décrits soit dans le langage *TTCN* ou *Menuet*. Pour déterminer les chemins à tester, l'outil utilise deux méthodes: l'évaluation symbolique ou l'analyse d'accessibilité. La première méthode pose beaucoup de restrictions sur les spécifications à tester. L'analyse d'accessibilité quant à elle ne pose aucune restriction sur la spécification, de plus, *Tveda V3* a comme interface un outil commercial très puissant d'analyse d'accessibilité appelé *Véda*.

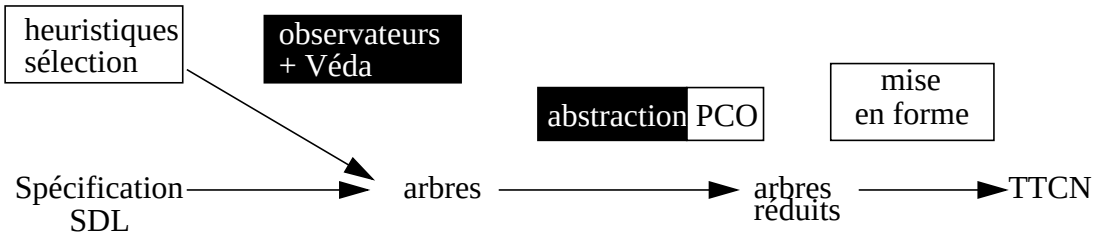



Figure 4.11 Schéma général de la méthode TVeda

### 4.13.4 TVeda V3+[Toua 96]

Cette méthode est une extension de la précédente et définit un but de test non pas comme une transition *SDL* mais comme un chemin du graphe syntaxique *SDL* partant de l'état initial de la spécification et se terminant soit par un *STOP* soit par un retour à l'état initial. Ainsi, les fonctionnalités du système décrites par le but de test correspondent à des comportements de bout en bout. Les cas de test sont automatiquement générés à partir des buts de test et d'une construction partielle du graphe d'accessibilité de la spécification *SDL*. Les buts de test générés correspondent à des séquences d'entrée/sortie et ne sont pas complets du fait qu'ils comportent des contraintes à résoudre, ce qui veut dire que les chemins de test associés ne sont pas exécutables et que l'usager devra les compléter.

### 4.13.5 TestGen [Anid 96]

*TestGen* optimise la méthode classique d'*UIO partielle* grâce à la construction d'un graphe d'accessibilité contraint et à l'application d'algorithmes de réduction à ce graphe. Cette méthode est basée sur une utilisation des méthodes classiques de test d'automates finis qui requièrent de disposer d'un automate fini calculable modélisant un protocole réel. La plupart du temps le graphe d'accessibilité de tels protocoles ne peut être calculé car il est trop gros. L'innovation de *TestGen* est de contraindre la génération de ce graphe en fixant les valeurs des paramètres des messages en entrée. Le graphe obtenu est de taille raisonnable. Aussi, *TestGen* résout le problème de la longueur des séquences de tests en utilisant un outil de réduction de graphe, *Aldébaran,* de la boîte à outil *CADP*, pour minimiser l'automate en fonction de l'architecture de test, avant d'engendrer les tests.

Cette méthode génère donc des cas de test incomplets car les valeurs des paramètres en entrées sont fixées dès le départ.

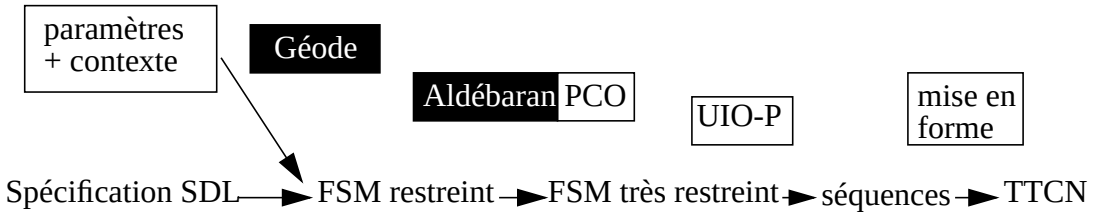



Figure 4.12 Schéma général de la méthode TestGen

### 4.13.6 TGV [Fern 96]

Dans cette méthode, plusieurs outils sont utilisés. L'outil commercial *Geode* (*VERILOG*) a servi à la génération du graphe d'état fini à partir de spécifications *SDL*. L'outil *Aldébaran*, introduit plus haut, a permis la minimisation de systèmes de transitions et un prototype appelé *TGV* (pour Test Generation using Verification techniques) et développé dans la boîte à outils *CADP* a permis la génération de suites de tests. *TGV* est fondé sur des techniques de vérification telles que le calcul de produit synchrone et la vérification à la volée.

### 4.13.7 TESDL [Brom 89]

*TESDL* est un prototype pour la génération automatique de cas de test à partir de spécifications *SDL*. Les spécifications *SDL* que cet outil peut manipuler ont certaines restrictions (un processus par bloc, deux processus ne peuvent recevoir le même genre de signal, etc). Les éléments *SDL* suivants sont supportés: canaux, blocs, un processus par bloc, les routes de signaux, état, entrée, sortie, tâche, décision, etc. L'outil accepte des spécifications *SDL* et produit des cas de test en *TTCN*.

### 4.13.8 TTCN Link [ITEX 1995]

*TTCN Link* (ou *LINK*) est un environnement pour le développement de suites de tests en *TTCN* à partir de spécifications *SDL* de l'ensemble d'outils *SDT 3.0* [SDT 1995]. *LINK* assure la consistance entre la spécification *SDL* et la suite de tests *TTCN*. Il augmente la productivité dans le développement en générant automatiquement la partie statique de la suite de tests. *LINK* est utilisé par le développeur de la suite de tests. Celui-ci a comme données d'entrée la spécification *SDL*, la structure de la suite de tests et il a pour tâche de développer une suite de tests *TTCN* abstraite.

### 4.13.9 TOPIC V2 [Alga 95]

*TOPIC V2* (prototype de *TTCGEN* (générateur automatique de cas de test pour *OBJECTGEODE: Verilog*), fait la co-simulation de la spécification *SDL* et d'un observateur représentant le but du test. Cette co-simulation permet d'explorer un graphe contraint, i.e., une partie du graphe d'accessibilité de la spécification, ce qui permet d'utiliser cette méthode pour les graphes infinis. L'observateur est décrit dans le langage *GOAL* (Geode Observation Automata Language). Afin de faciliter l'utilisation de *TOPIC*, il est possible de générer les observateurs à partir des *MSCs*. A partir du graphe contraint, des procédures sont exécutées pour générer les tests en *TTCN*.

Après ce survol des outil de génération de test existants, nous constatons que chaque outil contourne le problème de l'explosion combinatoire en imposant certaines restrictions sur la spécification ou bien en considérant un sous-ensemble de la spécification. Certains d'entre eux fixent les valeurs des paramètres en entrée, d'autres permettent de faire une simulation de la spécification par rapport à certains buts de test que l'usager doit entrer (sous forme de *MSCs* ou à l'aide d'un langage, e.g. *Goal*), ce qui permet de tester un sous-ensemble de la spécification seulement, et finalement d'autres posent des restrictions sur les constructions du langage de spécification.

## 4.14 Conclusion

Dans ce chapitre, nous avons passé en revue les méthodes de génération de cas de test existantes lorsque les systèmes sont modélisés par des *FSMs* et des *EFSMs*. Nous avons

également expliqué les avantages et les inconvénients de chacune d'elles. Par ailleurs, nous avons présenté les outils existants de génération de tests et nous nous sommes concentrés sur ceux qui génèrent des tests à partir de spécifications écrites dans le langage *SDL*. Nous présentons dans le prochain chapitre notre méthode de génération de cas de test pour les systèmes modélisés par des *EFSMs*. Cette dernière résout certains des problèmes des méthodes présentées dans ce chapitre.

# Chapitre 5

# La génération automatique de cas de test pour les systèmes modélisés par des EFSMs[1]

## 5.1 Introduction

La génération de cas de test dans le domaine des protocoles de communication, combinant les techniques de flux de contrôle et de données a été très bien étudiée. Dans le chapitre précédent, nous avons passé en revue les méthodes existantes de génération de cas de test pour les systèmes modélisés par des *FSMs* et des *EFSMs*. Dans [Chan 93], les auteurs présentent une méthode pour la génération automatique de cas de test, mais plusieurs cas de test exécutables ne sont pas généré. Cette dérnière utilise l'évaluation symbolique pour déterminer le nombre de fois qu'une boucle influente (à une transition) va être exécutée. Une transition influente est une transition qui modifie une ou plusieurs variables ayant une influence sur le flux de contrôle. Les variables sont appelées des variables influentes. [Ural 91] ne garantit pas l'exécutabilité des cas de test générés du fait qu'il ne considère pas les prédicats associés à chaque transition. De plus, le flux de contrôle n'est pas couvert. [Huan 95] génère des cas de test exécutables pour les protocoles modélisés par des *EFSMs* en utilisant l'analyse de flux de données. Pour remédier au problème de l'exécutabilité des séquences de tests, cette méthode utilise une traversée en largeur pour construire le graphe de la spécification, selon les données

1. Les résultats de ce chapitre sont publiés dans [Bour 97]

d'entrée lues et la configuration initiale. Cette méthode est un genre d'analyse d'accessibilité et possède donc les mêmes inconvénients notamment l'explosion combinatoire. De plus cette méthode ne considère pas le flux de contrôle.

## 5.2 Objectifs

Dans ce chapitre, nous allons présenter une méthode pour réaliser les tâches suivantes:

- utilisation d'une méthode unifiée pour tester le flux de contrôle ainsi que le flux de données de systèmes modélisés par des *EFSM*s;
- génération de cas de test exécutables; i.e., les prédicats associés à chaque transition d'un cas de test doivent être satisfaits;
- génération de cas de test pour les systèmes comportant des boucles non bornées; i.e, des boucles dont le nombre d'itérations dépend de la valeur de paramètres en entrée et utilisation d'une technique autre que l'analyse statique des boucles pour résoudre ce problème.
- [Chan 93] utilise l'analyse de boucles pour remédier au problème de l'exécutabilité. Si un chemin n'est pas exécutable, la méthode cherche dans le chemin s'il existe une boucle qui modifie la variable influente et l'ajoute autant de fois que nécessaire. Mais si le chemin ne comporte pas une telle boucle, il est rejeté. Notre méthode doit aussi être capable d'ajouter des bouts de chemins qui modifient la variable influente même si ce ne sont pas des boucles influentes. Afin de rendre exécutables les chemins non-exécutables, l'*analyse de cycles* est utilisée pour trouver le plus court cycle à insérer dans les chemins afin de les rendre exécutables. Un cycle est constitué d'une ou plusieurs transitions $t_1$,$t_2$,..,$t_k$ telles que l'état final de $t_k$ est le même que l'état initial de $t_1$.

## 5.3 Le modèle de fautes choisi

Le grand nombre et la complexité des fautes logicielles et matérielles incite qu'une approche pratique de test doit éviter de traiter directement ces fautes. Une méthode de

détection de la présence ou l'absence de fautes consiste à utiliser un modèle de fautes pour décrire les effets des fautes à un certain niveau d'abstraction (niveau logique, niveau RTL, blocs fonctionnels, etc). Ceci implique qu'il faut faire des concessions entre la facilité de modélisation et l'analyse. Si le modèle de fautes décrit les fautes d'une bonne façon, alors la seule tâche consiste à dériver les tests afin de détecter toutes les fautes du modèle de fautes. Cette approche a plusieurs avantages. Une faute de haut niveau peut causer plusieurs fautes physiques et logicielles, réduisant ainsi le nombre de possibilités à considérer lors de la génération de tests.

### 5.3.1 Méthode pour le test du flux de contrôle

Dans le chapitre précédent, nous avons montré les avantages de la méthode des *UIOs* par rapport aux autres méthodes. La méthode que nous avons choisie pour tester le flux de contrôle est donc celle des *UIOs*.

### 5.3.2 Méthode pour le test du flux de données

Notre but consiste à générer des cas de test pouvant tester le flux de contrôle ainsi que le flux de données des systèmes modélisés par des *EFSMs.* Pour ce qui est du test de flux de données, un choix doit être fait. Plus le critère est fort, plus le système est testé et vérifié dans le but de trouver des fautes. Cependant, un critère plus faible peut être réalisé, en général, en utilisant peu de cas de test.

Nous avons vu dans le chapitre 4 à la figure 4.8, la classification des critères de flux de données. Afin de pouvoir affirmer que nous testons le flux de données d'une manière efficace, le choix du critère le plus fort s'impose, i.e, "*tous-les-chemins*". Cependant, ce dernier devient trop cher et n'est plus pratique car les boucles de la spécification peuvent produire un nombre infini de chemins [Weyu 85]. Tester uniquement les chemins contenant des relations de flux de données est une approximation raisonnable, même si c'est plus faible que le critère "tous-les-chemins". Pour cette raison, nous choisissons le critère qui vient juste après soit le critère *définition-usages* ou *def-use*. En fait, nous allons nous concentrer sur la relation de dépendance entre les transitions. Deux types de dépendances sont définis entre deux transitions $t_i$ et $t_j$. $t_j$ est dépendante de $t_i$ par les

données (ou par le contrôle) s'il existe une variable x telle que: (i) $t_i$ contient une définition globale def de x, (ii) $t_j$ contient un c-usage global (ou p-usage de x) et (iii) il existe un chemin def-clear de $t_i$ à $t_j$ par rapport à x. Ainsi, pour tester le flux de données, nous allons générer tous les chemins def-use qui permettent de tester les dépendances qui existent dans le système entre les transitions (dépendances de données et de contrôle).

Nous devons souligner la différence entre le critère *def-use* et le modèle de fautes. Dans le cas du critère *def-use*, l'objectif est de satisfaire le critère en générant des cas de test qui testent les chemins correspondant au critère. Tester ces chemins ne garantit pas la détection des fautes existantes à cause des valeurs des variables qui doivent être choisies. Dans le cas où les bonnes valeurs sont choisies, le critère *def-use* est comparable à un modèle de fautes.

Maintenant que nous avons expliqué le modèle de fautes utilisé par notre méthode, nous présentons notre algorithme de génération de cas de test.

## 5.4 Algorithme de génération de cas de test

L'algorithme ci-dessous illustre le processus de génération automatique de cas de test à partir d'une spécification initiale du système à tester sous forme de *EFSM*. Ce dernier prend la spécification sous forme normale comme donnée d'entrée et produit comme données de sortie les cas de test complets exécutables ainsi que les séquences d'entrée/ sortie. Tout d'abord, le graphe de flux de données (*GFD*) est généré. A partir de ce dernier, les chemins def-use sont générés, puis les chemins qui sont inclus dans d'autres sont enlevés et les chemins restants sont complétés par des postambules. Comme les chemins def-use peuvent ne pas couvrir toutes les transitions de la spécification, des chemins additionnels peuvent être générés pour couvrir les transitions manquantes. Finalement, une séquence d'entrée/sortie est générée pour chaque cas de test exécutable.

**Algorithme *EFTG*** ***(Extended Fsm Test Generation)***

**Entrée**: spécification du système dans la forme normale

**Sortie**: Cas de test et séquences de test

**Début**

Lire la spécification de départ (*EFSM*)

Générer le graphe de flux de données G à partir de l'*EFSM*

Choisir une valeur pour chaque paramètre en entrée influençant le flux de contrôle

Génération-Chemins-def-use-Executables(G)

Enlever les chemins inclus dans d'autres

Ajouter l'identification d'état à chaque chemin def-use exécutable

Ajouter un postambule à chaque chemin def-use pour avoir un chemin complet

**Pour** chaque chemin complet **Faire**

Re-vérifier son exécutabilité

**Si** le chemin est encore non-exécutable **Alors**

Essayer de le rendre exécutable

**Si** le chemin est toujours non-exécutable **Alors**

le rejeter

**FinSi**

**FinSi**

**FinPour**

**Pour** chaque transition T non couverte par les cas de test générés **Faire**

Construire un chemin qui la couvre (pour le flux de contrôle).

**FinPour**

**Pour** chaque chemin exécutable **Faire**

Générer sa séquence d'entrée/sorties en utilisant l'évaluation symbolique

**FinPour**

**Fin;**

Dans ce qui suit, nous présentons la procédure qui génère les chemins def-use exécutables. Celle-ci fait appel à une autre procédure Trouver-Tous-Les-Chemins qui sera présentée dans la section 5.5.

La procédure de génération des chemins exécutables commence par la génération des usages de chaque transition de la spécification (A-Usage, I-Usage, C-Usage, P-Usage). Aussi le plus court préambule exécutable pour atteindre chaque état de la spécification est généré. Le but de cette procédure est de générer tous les chemins def-use; pour ce faire, la procédure trouve *tous les chemins* entre toute transition contenant une définition d'une variable v et toute autre transition contenant un usage de la même variable.

***Procédure Génération-Chemins-def-use-Executables***(GFD G)

**Entrée**: Le graphe de flux de données du système

**Sortie**: Les chemins def-use

**Début**

Générer les ensembles A-Usages, I-Usages, C-Usages et P-Usages pour chaque transition dans G

Générer le plus court préambule exécutable pour chaque état du graphe G;

**Pour** chaque transition T de G **Faire**

**Pour** chaque variable v ayant un A-Usage dans T **Faire**

**Pour** chaque transition U qui a un P-Usage ou un C-Usage de v **Faire**

Trouver-Tous-Les-Chemins(T,U)

**FinPour**

**FinPour**

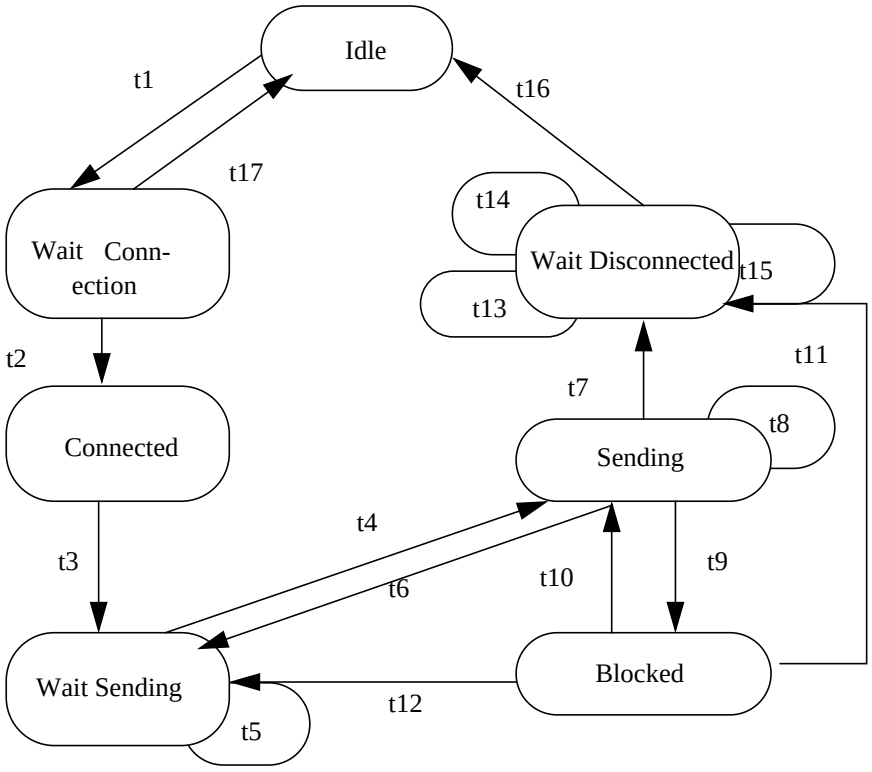
**FinPour**

**Fin**.

Nous tenons à préciser que nous sommes en train de générer des chemins entre deux transitions et non pas entre deux états (ce qui est habituellement réalisé). Pour générer un chemin entre deux transitions T1 et T2, nous générons en fait un chemin entre l'état final de T1 et l'état initial de T2, auquel on ajoute au début T1 et à la fin T2. Ceci est dû au fait que nous cherchons des chemins où une variable est définie dans la première transition et où elle est utilisée dans la dernière.

Dans la figure 5.1, nous présentons l'*EFSM* correspondant à une version simplifiée d'un vrai protocole. "? Msg" représente un message en entrée pour la transition en question, alors que "! Msg" représente un message en sortie. Cette notation est couramment utilisée pour spécifier les messages en entrée et en sortie pour les protocoles de communication.

Nous devons aussi signaler l'importance de certaines transitions dans l'exemple présenté dans la figure 5.1. Étant donné que les paramètres n et b ont une influence sur le flux de contrôle (i.e, leur valeur peuvent influencer le nombre de fois que certaines transitions telles que t4, t8 ou t9 vont être exécutées), les transitons t4, t8 et t9 sont appelées des **transitions influentes**. De même, la transition t8 est une **boucle influente** puisque c'est une transition influente qui est aussi une boucle.




t1 ?U.sendrequest
! L.cr

t2?L.cc
! U.sendconfirm

t3 ?U.datarequest(sdu, n,b)
number:=0;
counter:=0;
no_of_segment:=n;
blockbound:=b;

t4 ?L.tokengive
! L.dt(sdu[number])
start timer
number:=number+1;

t5 ?L/resume

t6 expire_timer
! L.tokenrelease

t7 ?l.ack() and number==no_of_segment
!U.monitor_complete(counter)
!token_release
!L.disrequest

t8 ?L.ack() number<no_of_segment and not expire_timer
!L.dt(sdu[number])
number:=number+1

t9 ?L.block not expire_timer
counter:=counter+1

t10 ?L.resume not expire_timer and counter<=blockbound

t11 counter>blockbound
!L.token_realease
!U.monitor_incomplete(number)
!U.dis_request

t12 expire_timer and counter<=blockbound
!L.token_release

t13 ?L.resume

t14 ?L.block

t15 ?L.ack

t16 ?L.dis_request
!U.disindication

t17 ?L.disrequest
!U.disindication

Figure 5.1 Exemple d'un protocole spécifié par une EFSM.

Supposons que la variable n est égale à 3 et qu'on est en train de voir si un chemin contenant la transition t7 est exécutable. Ce dernier n'est exécutable que si le prédicat de la transition t7 (number = number_of_segment) est satisfait (i.e., number est égal à 3). En d'autres termes, les transitions t4 ou t8 doivent apparaître 3 fois dans le chemin puisque ce sont les transitions qui incrémentent la variable **number**. Les variables **number** et **counter** sont appelées des variables influentes.

La table 9 présente les ensembles I-Usage, A-Usage et C-Usage pour les transitions de l'*EFSM* de la figure 5.1 ayant des usages.

| Transition | I-Usage | A-Usage | C-Usage | P-Usage |
|---|---|---|---|---|
| t3 | sdu, n, b | number, counter, no_of_segment, blockbound | n, b | - |
| t4 | - | number | number | - |
| t7 | - | - | counter | number, no_of_segment |
| t8 | - | number | number | -number, no_of_segment |
| t9 | - | counter | counter | - |
| t10 | - | - | - | counter, blockbound |
| t11 | - | - | number | counter, blockbound |
| t12 | - | - | - | counter, blockbound |

**TABLE 9. Ensemble d'usages pour certaines transitions de l'EFSM de la figure 5.1**

La raison pour laquelle nous cherchons un préambule pour atteindre un état est la suivante: supposons que nous voulons chercher tous les chemins def-use entre t3 et t7. Puisque l'état de départ de t3 n'est pas l'état initial du système, alors, tout chemin entre t3 et t7 ne peut devenir exécutable que si un préambule pour atteindre t3 lui est attaché. Ceci permettra d'interpréter les variables se trouvant dans les prédicats du chemin. De plus, le préambule à attacher à t3 doit aussi être exécutable. Supposons que nous voulons générer un préambule pour atteindre la transition t7 et que le paramètre en entrée n est égal à 2. Le plus court chemin serait alors t1, t2, t3, t4. Cependant, le chemin t1, t2, t3, t4, t7 n'est pas exécutable du fait que le prédicat de la transition t7 n'est pas satisfait. Après l'interprétation du predicat, l'expression "number = no_of_segment" devient "1=2", ce qui n'est pas vrai. Afin que le prédicat soit satisfait, la variable influente number doit être

incrémentée une deuxième fois avant la transition t7. Pour cela, une transition influente (t4 ou t8) permettant d'incrémenter la variable number doit être ajoutée avant t7. Deux possibilités sont considérées, notamment le cycle influent (t6, t4) à insérer après la transition t4 du chemin, ou bien la boucle influente t8 et t8 est choisie du fait que c'est le cycle le plus court. Le préambule exécutable pour t7 devient donc t1, t2, t3, t4, **t8**, t7.

| Transition | Préambule exécutable |
|---|---|
| t2 | t1 |
| t3 | t1, t2 |
| t4 | t1, t2, t3 |
| t5 | t1, t2, t3 |
| t6 | t1, t2, t3, t4 |
| t7 | t1, t2, t3, t4, t8 |
| t8 | t1, t2, t3, t4 |
| t9 | t1, t2, t3, t4 |
| t10 | t1, t2, t3, t4, t9 |
| t11 | t1, t2, t3, t4, t9, t10, t9, t10, t9 |
| t12 | t1, t2, t3, t4, t9 |
| t13 | t1, t2, t3, t4, t8, t7 |
| t14 | t1, t2, t3, t4, t8, t7 |
| t15 | t1, t2, t3, t4, t8, t7 |
| t16 | t1, t2, t3, t4, t8, t7 |
| t17 | t1 |

**TABLE 10. Préambule exécutable pour chaque transition de l'EFSM de la figure 5.1**

Dans la table 10, nous présentons le plus court préambule exécutable pour chaque transition (ou si on veut, pour atteindre l'état initial de chaque transition). Il faut signaler que les paramètres en entrée n et b ont tous les deux la valeur 2. Lors de la détermination du préambule pour chaque état, nous essayons de trouver le plus court chemin, pour atteindre un état, qui ne contient pas de prédicats. Si un tel chemin n'existe pas, nous cherchons alors le plus court chemin, que nous essayons éventuellement de rendre exécutable.

## 5.5 Génération des chemins def-use exécutables

Dans [Chan 93], après avoir ajouté les préambules et postambules aux chemins def-

use, leur exécutabilité est vérifiée. Cependant, plusieurs chemins restent non-exécutables et sont rejetés car des prédicats associés à ces chemins sont faux (non satisfaits). Afin de remédier à ce problème, nous vérifions l'exécutabilité de chaque chemin durant le processus de génération des chemins def-use. Nous présentons ci-dessous l'algorithme qui trouve tous les chemins entre deux transitions T1 et T2. Tout d'abord, nous aimerions mentionner que la procédure qui suit peut être utilisée pour deux choses: (i) générer tous les chemins def-use entre deux transitions T1 et T2, (ii) ou bien générer un cycle, dont l'état initial est l'état final de T1 et dont l'état final est l'état initial de T2, qui doit être inséré dans un chemin afin de le rendre exécutable. Dans le premier cas, un préambule doit être attaché au chemin généré (si l'état de départ de la transition T1 n'est pas l'état initial de la spécification). Dans le deuxième cas, aucun préambule n'est requis.

**Procédure *Trouver-Tous-Les-Chemins****(T1,T2, var)*
**Entrée**: Deux transitions T1 et T2 ainsi qu'une variable de contexte var
**Sortie**: Tous les chemins def-use entre T1 et T2 par rapport à la variable var
**Début**
  **Si** un cycle est à générer **Alors**
    Preambule:=T1
  **Sinon**
    Préambule:= le plus court préambule exécutable pour atteindre T1
  **FinSi**
  Générer-Tous-Les-Chemins(T1,T2,first-transition, var, préambule)
**Fin**;

Si nous voulons trouver tous les chemins def-use entre deux transitions T1 et T2, le préambule utilisé est le plus court préambule pour atteindre l'état initial de T1. Par contre, si notre but est de trouver un cycle, alors le préambule utilisé est constitué de T1.

Le prochain algorithme est l'algorithme utilisé pour trouver tous les préambules exécutables et tous les chemins def-use exécutables entre les transitions T1 et T2 par rapport à la variable var définie dans T1. Dans le cas où cette procédure est appelée pour générer tous les chemins def-use entre deux transitions, celle-ci retourne tous les chemins def-use tels que la première transition de chaque chemin contient une définition de la variable var et la dernière transition contient un usage (p-usage ou c-usage) de la même

variable. La variable var ne peut être redéfinie dans une transition autre que T1. Ainsi, avant d'ajouter une nouvelle transition T au chemin, nous vérifions si elle ne contient pas de re-définition de la variable var. Dans la négative, la transition T est ajoutée au chemin et la procédure Générer-Tous-Les-Chemins est appelée d'une manière récursive afin de trouver tous les chemins entre T et T2. Aussi, après la génération d'un chemin def-use entre T1 et T2, la procédure essaie de le rendre exécutable s'il ne l'est pas. En rendant un chemin exécutable, il peut devenir identique à un chemin déjà généré. Dans ce cas, le chemin est éliminé.

**Procédure *Générer-Tous-Les-Chemins****(T1, T2, T, var, Préambule)*
**Entrée**: Deux transitions T1, T2, une variable de contexte var et un Préambule
**Sortie**: Tous les chemins def-use entre T1 et T2
**Début**
**Si** (T est un successeur immédiat de T1) (e.g. t3 est un successeur immédiat de t2) **Alors**
**Si** (T=T2 ou (T suit T1 et T2 suit T dans G)) (e.g. t4 suit t2) **Alors**
**Si** nous sommes en train de construire un nouveau chemin **Alors**
Précédent:= le dernier chemin def-use généré (sans son préambule)
**Si** (T1 existe dans le chemin précédent) **Alors**
Commun:= la séquence de transitions dans Précédent se trouvant avant T1
**FinSi**
**FinSi**
**Si** nous sommes en train de construire un nouveau chemin **Alors**
Ajouter Préambule à Chemin
Ajouter var dans liste des buts de test pour Chemin
**FinSi**
**Si** Commun est non vide **Alors**
Ajouter Commun à Chemin
**FinSi**
**Si** (T = T2) **Alors**
Ajouter T à Chemin
Rendre-Executable(Chemin)
**FinSi**
**Sinon**
**Si** T n'est pas présente dans Chemin (mais peut être présente dans Préambule) et T ne possède pas un A-Use de var **Alors**
Ajouter T à Chemin

Générer-Tous-Les-Chemins(T, T2, première-transition, var, Préambule)
**FinSi**
**FinSi**
**FinSi**
T:= transition suivante dans le graphe
**Si** (T est non nulle) **Alors**
Générer-Tous-Les-Chemins(T1, T2, T, var, Préambule)
**Sinon**
**Si** (Chemin est non vide) **Alors**
**Si** (la dernière transition dans Chemin n'est pas un successeur immédiat de T2) **Alors**
Enlever la dernière transition de Chemin
**Sinon**
**Si** (Chemin est ou sera identique à un autre chemin après l'ajout de T2) **Alors**
Rejeter Chemin
**FinSi**
**FinSi**
**FinSi**
**FinSi**
**Fin**.

L'algorithme utilisé pour trouver tous les cycles est similaire, excepté qu'il n'appelle pas la procédure Rendre-Executable(chemin). Celle-ci n'est appelée que pour les chemins def-use. La procédure Rendre-Executable(P1) trouve la transition non exécutable dans P1, si elle existe. Si P1 = $(t_1,t_2,..t_{k-1},t_k)$ et si $t_k$ est la transition non exécutable, alors la procédure vérifie si un autre chemin def-use exécutable P2 = $(t_1,t_2,..,t_{k-1},...,t_k)$ existe. Si tel est le cas, P1 est rejeté. Sinon, la procédure Exécutabilité(P1) est appelée (voir section suivante). Cette vérification permet de gagner du temps en évitant de générer le même chemin ou un chemin équivalent (le même chemin avec un cycle différent) plus d'une fois.

Pour la génération des préambules et des postambules, nous avons implanté un algorithme qui cherche les plus courts préambules pour atteindre tous les états du système. Ce dernier se base sur l'algorithme Dijkstra [Dijk 59] et permet aussi de générer tous les plus courts postambules.

La table 11 montre tous les chemins def-use (avec le préambule ($t_1$, $t_2$, $t_3$, $t_4$)) de $t_9$ à $t_7$ par rapport à la variable *counter* ainsi que la raison pour laquelle quelques chemins ont été rejetés. Tous les chemins rejetés quand le prédicat est devenu (3=2) ne peuvent être rendus exécutables du fait que la transition influente apparaît plus qu'il ne le faut. Ceci montre que notre algorithme n'essaie pas de rendre certains chemins exécutables s'il se rend compte qu'il seront équivalents à des chemins qui sont déjà générés (e.g., le chemin t1, t2, t3, t4, t9, t10, t7 n'est pas exécutable et sera équivalent au premier chemin de la table après l'avoir rendu exécutable).

Dans la table suivante, P désigne le préambule pour atteindre t9; (P = t1, t2, t3, t4).

| Chemin Def-use | Rejeté | Raison du rejet |
|---|---|---|
| P, t9, t10, t6, t4, t7 | non | - |
| P, t9, t10, t6, t4, t8, t7 | oui | prédicat dans t7 devient (3=2) |
| P, t9, t10, t6, t5, t4, t7 | non | - |
| P, t9, t10, t6, t5, t4, t8, t7 | oui | prédicat dans t7 devient (3=2) |
| P, t9, t10, t7 | oui | sera équivalent au premier chemin après l'avoir rendu exécutable |
| P, t9, t10, t8, t6, t4, t7 | oui | prédicat dans t7 devient (3=2) |
| P, t9, t10, t8, t6, t5, t4, t7 | oui | prédicat dans t7 devient (3=2) |
| P, t9, t10, t8, t7 | non | - |
| P, t9, t12, t4, t7 | non | - |
| P, t9, t12, t4, t8, t7 | oui | prédicat dans t7 devient (3=2) |
| P, t9, t12, t5, t4, t7 | non | - |
| P, t9, t12, t5, t4, t8, t7 | oui | prédicat dans t7 devient (3=2) |

**TABLE 11. Chemins exécutables de t9 à t7 par rapport à la variable counter**

La procédure Exécutabilité(Chemin) vérifie d'abord si chaque transition dans Chemin est exécutable ou pas. Ceci est réalisé en interprétant chaque prédicat dans chaque transition symboliquement jusqu'à ce que ce dernier contienne uniquement des constantes et/ou des paramètres en entrée. Si une variable x est définie en fonction d'autres variables, alors on remonte en arrière jusqu'à ce que leur valeur, définie uniquement en fonction de constantes et/ou de paramètres en entrée soit trouvée, nous remplaçons alors x par ces

définitions. Puisque notre outil choisit les valeurs appropriées pour les paramètres en entrée qui ont une influence sur le flux de contrôle, tous les prédicats peuvent être interprétés. La raison est la suivante: pour la plupart des protocoles de communication, les contraintes sont très simples, ce qui permet aux prédicats d'être évalués et à l'algorithme de déterminer si la transition contenant le prédicat est exécutable ou pas. Cependant, pour certaines spécifications avec des boucles non bornées, la procédure Exécutabilité peut ne pas réussir à rendre un chemin exécutable.

Dans la prochaine section, nous définissons l'analyse de cycles et nous montrons comment elle peut être utilisée pour rendre un chemin exécutable.

## 5.6 Comment transformer des chemins non-exécutables en chemins exécutables?

Le problème de l'exécutabilité est en général non décidable. Cependant, dans la plupart des cas, il peut être résolu. [Rama 95] s'intéresse surtout à l'exécutabilité des préambules et postambules et ne s'occupe pas de l'exécutabilité des chemins couvrant le critère du flux de données. [Huan 95] a surmonté ce problème en exécutant l'*EFSM* afin de trouver tous les chemins exécutables. Le problème de cette méthode est que les chemins générés ne couvrent pas le flux de contrôle. De plus, cette méthode ne peut pas traiter de grandes *EFSM*s car elle essaie de trouver toutes les configurations possibles un peu comme l'analyse d'accessibilité et donc est sujette au problème de l'explosion combinatoire. Dans [Chan 93] les auteurs utilisent l'analyse statique et l'évaluation symbolique pour déterminer combien de fois une transition qui est en même temps une boucle, boucle influente, doit être exécutée pour que le cas de test devienne exécutable. Cette méthode n'est pas appropriée pour les spécifications où la variable influente n'est pas mise à jour à l'intérieur d'une *boucle* influente telle la variable influente **counter** qui est incrémentée à l'intérieur de la transition t9, dans la spécification dans la figure 5.1. Ainsi, [Chan 93] ne peut générer aucun cas de test exécutable couvrant la transition t11, puisque pour que t11 soit exécutable, la variable **counter** doit être incrémentée 2 fois (si n est égal à 2). Or dans l'exemple de la figure 5.1, il n'y a aucune boucle influente qui incrémente la variable **counter**. Aussi, [Chan 93] ne peut être utilisée si le nombre d'itérations d'une

boucle n'est pas connu à l'avance. Nous allons tout d'abord présenter le problème rencontré lors du test des boucles dans des programmes.

Le test des boucles

Les boucles forment une grande partie de la plupart des algorithmes. Le test des boucles est une technique de test de boîte blanche. On peut définir quatre classes de boucles.

- Les boucles simples: les prochains ensembles de tests doivent être appliquées aux boucles simples, où n est le nombre maximum d'itérations permises.
    - Eviter de passer par la boucle
    - Passer une seule fois par la boucle
    - Passer deux fois par la boucle
    - Passer m fois par la boucle où m < n
    - Passer (n-1) fois, n fois et (n+1) fois par la boucle
- Les boucles imbriquées: en étendant l'approche des boucle simples aux boucles imbriquées, le nombre de tests possibles va augmenter géométriquement à mesure que le nombre de niveaux des boucles augmente. Beizer [Beiz 90] suggère une approche pour réduire le nombre de tests. Dans cette approche, on commence d'abord par la boucle la plus interne et on initialise toutes les autres boucles aux valeurs minimales. Un test de boucle simple est réalisé pour cette boucle et des tests de valeurs limites et de valeurs exclues sont ajoutés. Procéder de même pour les autres boucles jusqu'à ce que toutes les boucles ont été testées.
- Les boucles concatenées: ces boucles peuvent être testées en utilisant l'approche définie ci-dessus pour les boucles simples si chaque boucle est indépendante de l'autre. Cependant, si deux boucles sont concatenées et si le compteur de la

première boucle est utilisé comme valeur initiale de la deuxième boucle, les deux boucles ne sont pas indépendantes; l'approche appliquée aux boucles imbriquées est alors recommandée.

- Les boucles non-structurées: aussi souvent que possible, ces boucles doivent être re-écrites pour refléter l'usage des constructions de la programmation structurée.

Pour tester les boucles d'un système modélisé par une *EFSM*, la méthode utilisée n'est pas identique aux méthodes mentionnées ci-haut. Les cas de test générés doivent contenir le nombre exact d'itérations de la boucle afin que ces derniers soient exécutables.

La méthode utilisée dans [Chan 93] pour tester les boucles n'est pas suffisante. En effet, cette dernière rend les chemins exécutables en ajoutant la boucle influente, celle qui permet de modifier la variable apparaissant dans le prédicat non satisfait d'une transition du chemin, un certain nombre de fois. La boucle influente est une transition qui a le même état de départ et d'arrivée et qui incrémente ou décrémente la variable influente. Dans [Chan 93], si la variable influente est modifiée dans une transition qui n'est pas une boucle, alors le chemin est simplement rejeté. Afin de surmonter ce problème, les cycles influents doivent être identifiés. Pour ce faire, nous avons développé l'heuristique suivante pour trouver le cycle approprié à insérer dans un chemin afin qu'il devienne exécutable.

**Procédure *Exécutabilité****(Chemin P)*

**Entrée**: Chemin P

**Sortie**: Chemin P exécutable ou nil si P ne peut être rendu exécutable

**Début**

Cycle:= nul

Traiter-Chemin(P)

**Si** P est encore non exécutable **Alors**

Rejeter P

**FinSi**

**Fin**;

La prochaine procédure a comme donnée d'entrée un chemin P et comme donnée de sortie le même chemin où un cycle a été inséré un certain nombre de fois. Tout d'abord, la transition T causant la non-exécutabilité du chemin est identifiée et un cycle influent, modifiant la variable influente de la transition T est trouvé. C'est la procédure Trouver-Tous-Les-Chemins présentée dans la section 5.4 qui est utilisée. Si un cycle non vide est trouvé, il est inséré dans le chemin et on revérifie s'il modifie correctement la variable influente. Si tel est le cas, le cycle est inséré autant de fois que c'est nécessaire, i.e., jusqu'à ce que le predicat de la transition T soit satisfait. Sinon, la procédure cherche un autre cycle s'il existe et les mêmes opérations sont répétées.

***Procédure Traiter-Chemin****(Chemin P)*
**Entrée:** Chemin P
**Sortie:** Chemin P traité
**Début**
T:= Première transition dans le chemin P
**Tant que** (T est non nulle) **Faire**
**Si** (T est non exécutable) **Alors**
Cycle:= Extraire-Cycle(P,T)
**FinSi**
**Si** (Cycle est non vide) **Alors**
Nb-Essais:=0
**Tant que (**T est non exécutable ET Nb-Essais < Max-Essais) **Faire**
Soit Précédent la transition avant T dans le chemin P
Insérer Cycle dans le chemin P après Précédent
Interpréter et évaluer le chemin P en commençant à la première transition de Cycle pour voir si les prédicats sont satisfaits ou pas
Nb-Essais:= Nb-Essais +1
**Si** (une transition de Cycle n'est pas exécutable) **Alors**
Enlever Cycle de P
Trouver un autre cycle Cycle s'il existe
**Si** Cycle est non nul **Alors**
Nb-Essais=0;
**FinSi**
**FinTQ**

**Sinon**
Exit
**FinSi**
T:= Prochaine transition de P
**FinTQ**
**Fin**.

Nous aimerions mentionner que l'outil *EFTG* fait la différence entre deux genres de prédicats. Un prédicat binaire a la forme suivante: "var1 R var2", où R est un opérateur relationnel tel que "<" alors qu'un opérateur unaire peut être écrit sous la forme F(x), où F est une fonction booléenne telle que "Pair(x)" (voir figure 5.2). Nous présentons ci-dessous l'algorithme de la procédure qui trouve le plus court cycle qui modifie la variable influente tel que requis.

***Procédure Extraire-Cycle*** *(Chemin P, Transition T)*

**Entrée:** Le chemin P ainsi que la transition T non exécutable
**Sortie:** Le cycle influent ou u cycle vide
**Début**
**Si** la non-exécutabilité est due à un prédicat unaire **Alors**
Trouver, parmi les transitions précédant T dans G, une transition avec
le même prédicat mais avec une valeur différente
Générer le plus court cycle influent contenant cette transition
**Sinon**
Trouver la variable influente causant la non-exécutabilité
Trouver une transition parmi les transitions précédant T dans G contenant
la variable influente et la modifiant comme demandé
Générer le plus court cycle influent contenant cette transition
**FinSi**
**Si** aucune transition influente n'existe **Alors**
Retourner un cycle vide
**FinSi**
**Fin**.

Afin de générer les cycles, une procédure similaire à *Générer-Tous-Les-Chemins* est utilisée. Cependant, elle n'essaie pas de rendre le cycle exécutable. L'heuristique

*Exécutabilité* vérifie si chaque chemin non exécutable peut devenir exécutable. Pour ce faire, nous trouvons la première transition non exécutable T dans le chemin P. Deux cas peuvent survenir:

- Si la transition T ne peut être exécutée du fait qu'un prédicat unaire n'est pas satisfait (e.g. Pair(x) n'est pas vrai), nous trouvons une transition $t_k$, si elle existe, parmi les transitions précédant la transition T, qui possède le même prédicat avec une valeur différente. Un cycle influent contenant $t_k$ est généré s'il existe et est inséré dans le chemin P avant la transition T. La première transition du cycle doit suivre la transition précédant T dans le chemin initial et la dernière transition du cycle doit précéder T. Cette condition assure que le cycle puisse être inséré dans le chemin P.

- Si le prédicat n'est pas unaire, nous trouvons, à l'aide de l'évaluation symbolique, quelle est la variable causant la non-exécutabilité et nous déterminons si cette variable doit être augmentée ou diminuée pour que la transition T devienne exécutable. Cette variable doit être une variable influente et des transitions modifiant la variable doivent exister. Si tel n'est pas le cas, un cycle vide est retourné et le chemin est rejeté. Si la variable est une variable influente, nous cherchons, parmi les transitions précédant T, une transition $t_k$ qui modifie la variable proprement, générons un cycle contenant la variable et l'insérons dans le chemin. De plus, lors de l'analyse du prédicat se trouvant dans la transition non exécutable, nous vérifions si la variable doit être augmentée ou diminuée pour que la transition devienne exécutable et le cycle approprié est généré, i.e., si la variable doit être augmentée, nous cherchons un cycle qui augmente la variable et vice versa. Si le chemin n'est toujours pas exécutable, il est rejeté.

Afin d'illustrer l'heuristique, supposons que dans l'*EFSM* de la figure 5.1 les deux paramètres en entrée n et b ont la valeur 2. Le plus court préambule exécutable pour $t_{11}$ est ($t_1$, $t_2$, $t_3$, $t_4$, $t_9$, $t_{11}$). Mais $t_{11}$ n'est pas exécutable parce que son prédicat "counter > 2" devient "1 > 2" après son interprétation. Notre outil trouve que la variable influente est "counter". Il trouve aussi que parmi les transitions précédant $t_{11}$, $t_9$ est une transition influente qui peut être adéquate du fait qu'elle augmente la variable "counter". Un cycle influent contenant $t_9$ est généré notamment ($t_{10}$,$t_9$) et inséré deux fois après la transition $t_9$

ce qui rend le chemin exécutable. Le chemin devient alors ($t_1$, $t_2$, $t_3$, $t_4$, $t_9$, $t_{10}$, $t_9$, $t_{10}$, $t_9$, $t_{11}$).

La table 12 présente les cas de test exécutables finaux (sans identification d'états) générés par notre outil pour l'*EFSM* de la figure 5.1. Dans plusieurs cas, l'outil a dû chercher un cycle influent afin de rendre un chemin exécutable.

| Chemin | Cas de test Exécutable | But du test |
|---|---|---|
| 1 | t1, t2, t3, t5, t4, t8, t7, t16, | number, counter, no_of_segment |
| 2 | t1, t2, t3, t5, t4, t8, t9, t10, t7, t16 | number, counter, no_of_segment, blockbound |
| 3 | t1, t2, t3, t5, t4, t9, t10, t8, t7, t16 | number, counter, no_of_segment, blockbound |
| 4 | t1, t2, t3, t4, t8, t9, t10, t9, t10, t9, t11, t16 | number, counter, no_of_segment, blockbound |
| 5 | t1, t2, t3, t5, t4, t8, t9, t10, t9, t10, t9, t11, t16 | number, counter, no_of_segment, blockbound |
| 6 | t1, t2, t3, t5, t4, t9, t10, t9, t10, t9, t11, t16 | number, counter, blockbound |
| 7 | t1, t2, t3, t5, t4, t9, t12, t4, t7, t16 | number, counter, blockbound |
| 8 | t1, t2, t3, t4, t6, t4, t7, t16 | number, counter, no_of_segment |
| 9 | t1, t2, t3, t4, t6, t5, t4, t7, t16 | number |
| 10 | t1, t2, t3, t4, t8, t7, t16 | number, counter, no_of_segment |
| 11 | t1, t2, t3, t4, t8, t9, t10, t7, t16 | number, counter, no_of_segment, blockbound |
| 12 | t1, t2, t3, t4, t9, t10, t6, t4, t7, t16 | number, counter, no_of_segment, blockbound |
| 13 | t1, t2, t3, t4, t9, t10, t6, t5, t4, t7, t16 | number, counter, blockbound |
| 14 | t1, t2, t3, t4, t9, t10, t8, t7, t16 | number, counter, no_of_segment, blockbound |
| 15 | t1, t2, t3, t4, t9, t12, t4, t7, t16 | number, counter, blockbound |
| 16 | t1, t2, t3, t4, t9, t12, t5, t4, t7, t16 | number, counter, blockbound |
| 17 | t1, t2, t3, t4, t9, t10, t9, t10, t9, t11, t16 | number, counter, blockbound |
| 18 | t1, t2, t3, t4, t9, t10, t6, t4, t9, t10, t9, t11, 16 | number, counter, blockbound |

**TABLE 12. Cas de test exécutables pour l'EFSM de la figure 5.1**

| Chemin | Cas de test Exécutable | But du test |
|---|---|---|
| 19 | t1, t2, t3, t4, t9, t10, t6, t5, t4, t9, t10, t9, t11, t16 | number, counter, blockbound |
| 20 | t1, t2, t3, t4, t9, t10, t8, t6, t4, t9, t10, t9, t11, t16 | number, counter, no_of_segment, blockbound |
| 21 | t1, t2, t3, t4, t9, t10, t8, t6, t5, t4, t9, t10, t9, t11, t16 | number, counter, no_of_segment, blockbound |
| 22 | t1, t2, t3, t4, t9, t10, t8, t9, t10, t9, t11, t16 | number, counter, no_of_segment, blockbound |
| 23 | t1, t2, t3, t4, t9, t12, t4, t9, t10, t9, t11, t16 | number, counter, blockbound |
| 24 | t1, t2, t3, t4, t9, t12, t5, t4, t9, t10, t9, t11, t16 | number, counter, blockbound |
| 25 | t1, t2, t3, t4, t8, t7, t13, t14, t15, t16 | - |
| 28 | t1, t17 | - |

**TABLE 12. Cas de test exécutables pour l'EFSM de la figure 5.1**

Avec identification d'états, le premier cas de test exécutable est le suivant: ($t_1$, $t_2$, $t_3$, $t_5$, $t_4$, $t_8$, $t_7$, $\mathbf{t_{15}}$, $t_{16}$).

Les deux derniers chemins sont ajoutés pour couvrir les transitions $t_{13}$, $t_{14}$, $t_{15}$ et $t_{17}$ qui ne sont pas couvertes par les autres chemins (pour couvrir le flux de contrôle).

Nous remarquons que certaines transitions apparaissent dans plusieurs cas de test telles que les transitions t4, t5, t6, t7, t9, alors que d'autres apparaissent dans un seul cas de test (e.g., t17). La raison est la suivante: les premières transitions sont couvertes par des chemins def-use et nous savons que ce critère exige que tous les chemins entre une définition d'une variable et un usage de la même variable doivent être générés, ce qui entraîne que plusieurs chemins couvrant une même transition sont générés. Les transitions t13,..,t17 quant à elles ne sont pas couvertes par les chemins def-use, mais puisque nous devons générer des cas de test couvrant toutes les transitions de la spécification (afin de couvrir le flux de contrôle), nous avons généré des cas de test pour les couvrir également.

Finalement, les séquences d'entrée/sortie sont extraites à partir des cas de test exécutables et appliquées pour tester l'implantation. Pour les paramètres en sortie avec des variables (tel que le paramètre en sortie *dt*), l'évaluation symbolique est utilisée pour

déterminer la valeur de la variable *number* qui a un usage de sortie.

Voici la séquence d'entrée/sortie pour le premier chemin exécutable: "?sendrequest!cr?cc!send_confirm?data_request(sdu,2,2)?resume?token_give!dt(0)!token_release?token_give!dt(1)?ack!monitor_complete(0)!token_release!dis_request?dis_request!dis_indication".

## 5.7 Autre exemple

La figure 5.2 présente un exemple de *EFSM* avec des boucles non bornées. Chaque boucle est formée d'une transition et possède un prédicat unaire. Pour cet exemple, puisque les transitions $t_2$ et $t_3$ ne sont pas bornées (le nombre d'itérations de chaque boucle n'est pas connu), plusieurs méthodes de génération de cas de test pour les systèmes modélisés par des *EFSMs* ne peuvent générer aucun cas de test exécutable pour cet exemple [Chan 93, Rama 95].

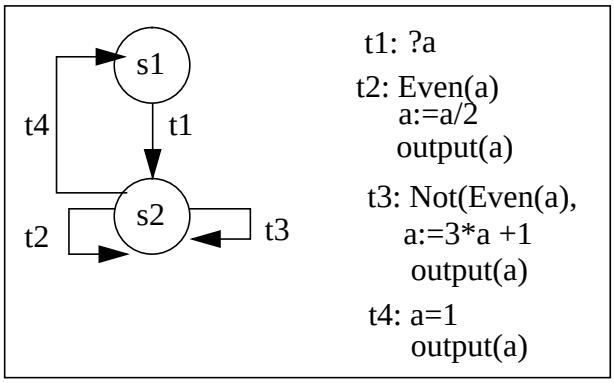



Figure 5.2 EFSM avec des boucles non bornées et des prédicats unaires.

Dans la table 13, les cas de test exécutables (sans identification d'états) ainsi que les séquences de tests pour l'*EFSM* de la figure 5.2 sont présentés. Chaque cas de test est

relatif à une valeur différente du paramètre en entrée *a*.

| Paramètre en entrée | Cas de test Exécutable | Séquence d'entrée/sortie |
|---|---|---|
| 1 | t1, t4 | ?1!1 |
| 5 | t1, t3, t2, t2, t2, t2, t4 | ?5! 16! 8! 4! 2! 1!1 |
| 100 | t1, t2, t2, t3, t2, t2, t3, t2, t3, t2, t2, t2, t3, t2, t3, t2, t2, t3, t2, t2, t2, t3, t2, t2, t2, t2, t4 | ?100!50!25!76!38!19!58!29 !88!44!22!11!34!17!52!26!1 3,!40, 20!10!5!16!8!4!2!1!1 |
| 125 | - | - |

**TABLE 13. Cas de test exécutables pour l'EFSM de la figure 5.2**

Pour l'*EFSM* de la figure 5.2, notre méthode a généré des cas de test pour plusieurs valeurs du paramètre en entrée et a échoué pour d'autres (e.g. 125). Ceci est dû au fait que dans la procédure Traiter-Chemin(P) de la section 5.5, nous avons fixé la valeur du paramètre Max-Essais. En augmentant la valeur de ce dernier, nous avons pu obtenir une solution pour une valeur du paramètre *a* égale à 125.

Après la génération des cas de test exécutables, une séquence d'entrée/sortie est générée pour chaque chemin. Les données d'entrée sont appliquées à l'implantation sous test et le sorties observées sont comparées aux sorties générées par l'outil *EFTG*.

## 5.8 Résultats

Afin de comparer notre outil à d'autres méthodes, nous avons implanté un algorithme qui génère tous les chemins def-use (comme dans [Chan 93]), auquel on a ajouté l'analyse de cycle pour traiter le problème de l'exécutabilité des cas de test au lieu de l'analyse de boucles. Appelons cet algorithme "Ch+". Notons que cet algorithme vérifie l'exécutabilité **après** que les chemins def-use soient générés.

Dans la table 14, nous résumons les résultats obtenus par Ch+ ainsi que ceux obtenus par notre outil pour 3 *EFSMs*. La troisième *EFSM* est une version simplifiée du protocole *INRES*. Elle a quatre états, quatorze transitions et quatre boucles dont deux sont des boucles influentes.

| **EFSM** | **Ch+** | | **Notre outil: EFTG** | | | |
|---|---|---|---|---|---|---|
| | Chemins | Exécutable | Chemins def-use | Chemins def-use rejetés | | Exécutable |
| figure 5.1 | 81 | 26 | 60 | 29 | 16 | 26 |
| figure 5.2 | 9 | 1 | 1 | 0 | 0 | 1 |
| INRES | 54 | 25 | 24 | 4 | 4 | 22 |

**TABLE 14. Comparaison des résultats obtenus par EFTG et par Ch+**

La première colonne des chemins rejetés par notre outil spécifie le nombre total de chemins rejetés durant la génération des chemins def-use. La seconde colonne spécifie le nombre de chemins qui sont rejetés par l'outil après avoir essayé de les rendre exécutables, du fait que des chemins équivalents existent déjà.

Nous pouvons donc conclure que *EFTG* a rejeté moins de chemins que *Ch+*. Ceci est dû au fait que *EFTG* vérifie l'executabilité d'un chemin au cours de sa génération; aussi, si *EFTG* s'aperçoit qu'en essayant de rendre un chemin exécutable, ce dernier va devenir équivalent à un chemin qui est déjà généré, le chemin est rejeté, ce qui permet d'éviter de perdre du temps. *EFTG* et *Ch+* diffèrent donc, non pas par le nombre de cas de test générés, mais plutôt par l'efficacité du processus de génération de cas de test.

## 5.9 Etude de la complexité

Soit E une *EFSM* et:

- t: le nombre de transitions de E,
- e: le nombre d'états de E,
- v: le nombre de variables de E,
- n: le maximum entre t et v.

Nous aimerions calculer la complexité de *EFTG*. Pour ce faire, nous devons calculer la complexité de la procédure "Génération-chemins-def-use-Executables(GFD G)".

Complexité de la procédure “Génération-chemins-def-use-Executables(GFD G)”:

Cette complexité est égale à: t * v * t * Complexité (Trouver-tous-les-chemins) qui est égale à $n^3$ * Complexité (Trouver-tous-les-chemins).

Complexité de “Trouver-tous-les-chemins”

Elle est égale à Complexité (Générer-tous-les-chemins(T1, T2,...)).

Complexité de “Générer-tous-les-chemins(T1, T2,...)”

La procédure Générer-tous-les-chemins(T1, T2,...) est très récursive. Quand une nouvelle transition T est ajoutée au chemin, la procédure est appelée récursivement afin de trouver tous les chemins entre la nouvelle transition et T2. Trouver ces chemins revient à appeler la procédure récursivement avec les paramètres (T, T2,...). Une fois que ces chemins sont trouvés, on revient en arrière afin de continuer la génération des chemins def-use en regardant si la transition suivant T (dans le graphe) peut être ajoutée au chemin. Du fait de la nature même de cette procédure, nous pouvons conclure que le temps de calcul est exponentiel. La complexité de cette procédure est donc $t^t$ ou bien $n^n$.

Complexité de la procédure Exécutabilité(P)

Elle est égale à la complexité de la procédure Traiter-Chemin(P) qui est égale à: t * (Complexité (Extraire-Cycle) + k * p); k étant le nombre de fois que le cycle est inséré et p est le temps pour insérer le cycle).

Complexité de Extraire-cycle

Elle est égale à la complexité de Générer le plus court cycle qui est égale à la complexité de Générer-tous-les-chemins.

La procédure Exécutabilité est aussi dans l’ordre de $n^n$.

Ainsi, nous pouvons conclure que la complexité de toutes les procédures mentionnées plus haut est la même et est égale à $n^n$.

En résumé, nous pouvons conclure que la complexité de *EFTG* dépend du critère de flux de données utilisé. Plus le critère est fort, plus l'algorithme de génération de cas de test est complexe. Pour cette raison, nous avons également implanté le critère "tous les usages" qui, contrairement au critère "toutes les définitions-usages" nécessite qu'un seul chemin entre une définition et un usage soit généré. En utilisant ce critère, la complexité de l'algorithme de génération de cas de test est fortement réduite.

Nous avons implanté l'algorithme de Floyd pour générer le plus court chemin entre deux états. La complexité de cet algorithme est $n^3$. La complexité de l'algorithme de génération des chemins "tous-les-usages" est donc dans l'ordre de $n^6$.

## 5.10 Conclusion

Comme nous l'avons mentionné auparavant, pour l'*EFSM* de la figure 5.1, 55 chemins sont rejetés par Ch+ (après la génération de tous les chemins def-use), alors que *EFTG* a rejeté seulement 29 (durant la génération des chemins def-use). Vérifier l'exécutabilité durant la génération des chemins def-use permet de générer seulement les chemins qui ont une forte probabilité d'être des chemins exécutables. Notre méthode génère les cas de test exécutables pour les systèmes modélisés par des *EFSM*s en utilisant les techniques d'évaluation symbolique pour évaluer les contraintes de chaque transition. De ce fait, seuls les cas de test ayant des contraintes satisfaites sont générés. Aussi, notre méthode découvre plus de cas de test exécutables que les autres méthodes car elle permet de trouver non seulement les boucles influentes mais les cycles influents aussi. Elle permet aussi de générer des cas de test pour des spécifications avec des boucles non bornées car elle n'utilise pas les techniques d'analyse statique. Finalement, après la génération des cas de test et des séquences de tests, les résultats peuvent être utilisés pour vérifier la conformité de l'implantation à sa spécification.

Nous devons cependant souligner les limitations de *EFTG*. Tout d'abord, la génération des chemins def-use devient rapidement très complexe à cause du critère de flux de données utilisé et l'implantation de critères de flux de données plus faibles serait très appréciable. Aussi, les prédicats ainsi que les instructions d'affectation comportent des opérateurs assez simples tels que la multiplication, la division, l'addition, la soustraction et

l’opérateur modulo et l’expression “a est pair” est transformée à “a%2 = 0” où “%” désigne l’opérateur modulo. En fait, pour les logiciels de communication, et du fait de leur simplicité, ceci ne constitue pas une limitation.

# Chapitre 6

# Génération de cas de test pour les systèmes modélisés par des CEFSMs[1]

## 6.1 Introduction

Dans le chapitre précédent, nous avons présenté notre méthode de génération automatique de cas de test pour les systèmes modélisés par une *EFSM*. Mais en réalité, la plupart des systèmes (ou protocoles de communication) sont modélisés par plusieurs machines ou processus qui communiquent via des files d'attente ou par rendez-vous. La méthode que nous avons présentée n'est donc plus utilisable pour ce genre de systèmes à moins de faire le produit cartésien (ou analyse d'accessibilité) de tous les processus composant le système et ensuite générer les cas de test pour ce produit en utilisant *EFTG*. Cependant, à cause du problème de l'explosion d'états, une autre solution doit être trouvée.

A notre connaissance, très peu de méthodes existent pour la génération de cas de test pour les systèmes modélisés par plusieurs modules et la plupart de ces méthodes génèrent les cas de test pour les systèmes modélisés par des *FSM* communicantes (*CFSMs*). La partie données du système n'est donc pas considérée.

1. Les résultats de ce chapitre sont publiés dans [Bour 98]

Une approche simple pour tester les *CFSM*s est d'en faire le produit cartésien (la composée) puis de générer des cas de test pour le produit. Mais cette approche possède un inconvénient majeur qui est l'explosion combinatoire. Aussi, l'utilisation de cette approche n'est pas convenable car elle ne tient pas compte des variables du système.

Afin de gérer cette complexité, des méthodes d'analyse d'accessibilité réduite ont été proposées pour les systèmes modélisés par des *CFSMs* [Rubi 82, Goud 84, Lee 96]. L'idée principale consiste à construire un plus petit graphe représentant un comportement partiel du système et permettant d'étudier les propriétés de communication.

Dans [Hwan 97], les auteurs proposent une méthode pour générer une seule *EFSM* à partir d'un système modélisé par plusieurs *EFSM*s pour le test du flux de données. Cette méthode considère aussi que les messages dans les files d'attente de chaque *EFSM* ne sont pas inclus dans les états globaux.

Dans ce chapitre, nous présentons une méthode qui peut être utilisée pour tester un système composé de plusieurs *EFSM*s communicantes (*CEFSM*s) ou bien une partie du système après une correction ou une augmentation du système.

Comparée aux méthodes d'analyse d'accessibilité réduite (comme celles présentées dans la section 2.6.3) qui considèrent toutes les transitions dans toutes les *EFSM*s puis choisissent, à l'aide d'une technique, les transitions à exécuter à chaque étape de l'analyse réduite, notre méthode choisit d'abord un sous-ensemble de transitions, ensuite l'analyse d'accessibilité est effectuée. Notre méthode ne compose pas toutes les *CEFSM*s, mais décompose le problème posé en calculant un produit partiel (défini plus tard) pour chaque *CEFSM* et en générant des cas de test pour le produit partiel en utilisant l'outil *EFTG* présenté dans le chapitre précédent. Pour calculer le produit partiel pour une certaine *CEFSM*, nous calculons le produit de cette *CEFSM* avec les parties (sous-ensembles de transitions) d'autres CEFSMs qui influencent cette *CEFSM* ou qui sont influencées par elle. En d'autres termes, nous considérons toutes les transitions dans la *CEFSM* et un sous-ensemble des transitions de chaque autre *CEFSM* qui est impliquée dans la communication avec la machine pour laquelle nous désirons calculer le produit partiel. Nous l'appelons *produit partiel* du fait que toutes les transitions ne sont pas considérées

lors de l'analyse d'accessibilité. Ainsi, nous résolvons le problème de la génération de cas de test pour un système modélisé par plusieurs *CEFSM*s en testant chaque *CEFSM* dans son contexte. La complexité de la génération de cas de test pour le système global est donc considérablement diminuée ce qui permet de tester des systèmes réels. Dans la suite, nous utilisons le terme *CEFSM dans le contexte* pour designer le produit partiel de la *CEFSM*.

Notre objectif n'est pas de couvrir toutes les transitions globales (issues du produit de toutes les *CEFSMs*), mais de couvrir les transitions de chaque *CEFSM*, toutes les transitions globales de chaque produit partiel ainsi que les chemins def-use (de flux de données) pour chaque *produit partiel*.

Ce chapitre est organisé comme suit. Dans la section 6.2, nous définissons ce qu'est un système communicant. La troisième section présente un algorithme qui calcule le produit partiel pour une *CEFSM*. La section 6.4 présente un algorithme pour générer des cas de test pour un système modélisé par plusieurs *CEFSM*s. Dans la section 6.5, nous proposons une méthode pour réduire la taille du produit partiel ou total. Un exemple est présenté dans la section 6.6. L'analyse des résultats est présentée dans la section 6.7 et la section 6.8 conclut ce chapitre.

## 6.2 Système communicant

**Définition 6.1**: Un système communiquant $\Pi$ de k *CEFSMs* est un 2*k tuple ($C_1$, $C_2$,..., $C_k$, $F_1$, $F_2$,..., $F_k$) où

- $C_i$ = <S, $s_0$, I, O, T, A, $\delta$ , V> est une *EFSM* telle que définie dans le chapitre 4.
- $F_i$ est une file d'attente *FIFO* (First In First Out) pour $C_i$, i = 1..k.

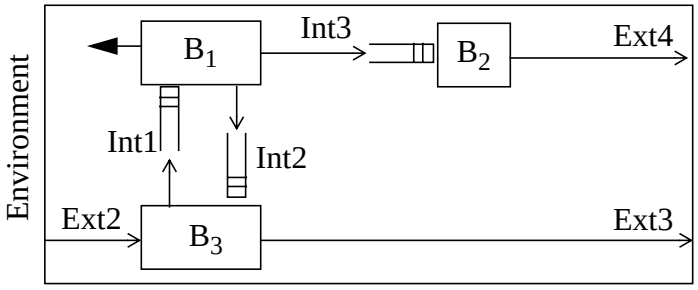
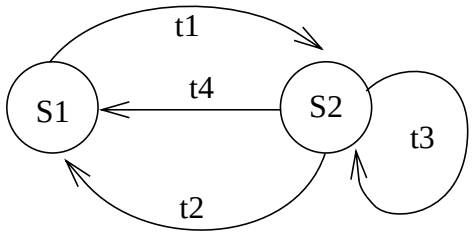



Figure 6.1 Exemple de système communicant

La figure 6.1 présente un exemple de système communicant, alors que la figure 6.2 présente le détail sur chaque *CEFSM* du système global.



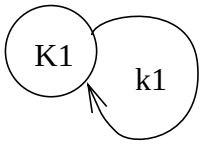

(a) B1

t1: ?Ext1(c)
R:=c*c;

t2: ?int1
!Int2(R);

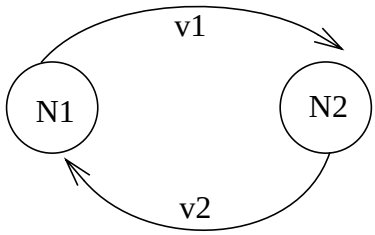
t3: (R is odd)
R:=R+1;

t4: (R is even)
L:=R/2;
!Int3(L);




(b) B2

k1: ?Int3(L)
!Ext4(L);




(c) B3

v1: ? Ext2
!Int1

v2: ?Int2(R)
!Ext3(R)

Figure 6.2 Un système de 3 CEFSMs: B1, B2 et B3

Considérons l'exemple de la figure 6.2. Nous avons 3 processus B1, B2 et B3 qui communiquent l'un avec l'autre ainsi qu'avec l'environnement.

B1 lit une donnée externe Ext1 avec le paramètre c et affecte c*c à R dans t1. A l'état s2, si B1 reçoit la donnée d'entrée Int1 de B3, il répond en envoyant Int2 contenant la valeur de R. R peut être modifie dans t3 avant d'être envoyé à B3.

Dans t4, si R est pair, le message Int3 est envoyé à B2 avec la valeur calculée de L.

Ext veut dire un message reçu ou envoyé à l'environnement alors que Int signifie que c'est un message interne (entre blocs). Aussi, "?msg" représente une donnée en entrée alors que "!msg" représente une donnée en sortie.

## 6.3 Génération guidée de cas de test pour une CEFSM

Le fait de tester une *CEFSM* C en isolation réduit le coût du test. Cependant, le problème majeur dans le test de la machine C réside dans son interaction avec les autres *CEFSM*s. Ceci est dû au fait que les cas de test générés pour C en isolation ne considèrent pas l'interaction de C avec les autres machines. Aussi quelques cas de test exécutables de C peuvent ne plus l'être lorsque la communication de C avec les autres machines est prise en compte. Ainsi, puisque le fait de tester chaque *CEFSM* séparément n'est plus suffisant et puisque la composition de toutes les machines coûte très cher, nous allons utiliser une approche milieu.

Nous allons tester un système communicant en testant chacune de ses *CEFSMs* dans son contexte. À notre connaissance, il n'existe aucune méthode de génération de cas de test pour les *CEFSM*s qui utilise ce principe.

Dans ce qui suit, nous générons un produit partiel pour chaque *CEFSM* et nous utilisons l'outil *EFTG*, présenté dans le chapitre précédent, pour générer des cas de test pour le produit partiel, étant donné que le produit partiel est une *EFSM*. En général, afin de tester un système modélisé par plusieurs *CEFSMs*, chaque *CEFSM* est testée en isolation afin de s'assurer qu'elle est correcte. Ensuite, le système global est testé. Le prochain algorithme calcule le produit partiel pour une *CEFSM* et génère des cas de test pour le

produit partiel à l'aide de *EFTG*. En plus, l'algorithme utilise, comme guide, les cas de test générés par *EFTG* pour tester chaque *CEFSM* en isolation et pour calculer le produit partiel. Dans ce cas, les cas de test générés pour le produit partiel couvrent les critères de flux de contrôle et de flux de données utilisés par *EFTG*. Nous avons donné à notre méthode le nom de *procédure guidée* car elle utilise les cas de test générés par *EFTG* pour chaque machine en isolation comme guide pour choisir les transitions qui vont participer à la génération du produit partiel.

Le processus de génération du produit partiel pour la *CEFSM* $C_n$ est divisé en quatre étapes:

- Génération de cas de test pour toutes les *CEFSMs* en isolation (cette étape est effectuée une seule fois)
- Marquage de $C_n$ et de toutes les transitions dans les autres machines qui ont une influence sur $C_n$ ou bien qui sont influencées par $C_n$
- Génération du produit partiel pour $C_n$, soit $PP(C_n)$
- Génération des cas de test pour $PP(C_n)$ en utilisant le critère def-use.

Nous allons utiliser l'exemple de la figure 6.2 pour illustrer chacune des étapes en prenant $C_n$ = B3.

**Etape 1: La génération des cas de test pour toutes les CEFSMs en isolation.**

Cette étape consiste à appeler *EFTG* pour générer des cas de test exécutables pour chaque *CEFSM* en isolation. Les cas de test générés pour chaque machine sont des cas de test exécutables qui couvrent toutes les transitions de la machine ainsi que tous ses chemins def-use.

La table 15 présente les cas de test exécutables générés par *EFTG* pour chaque machine en isolation.

| Cas de test pour B1 | Cas de test pour B2 | Cas de test pour B3 |
|---|---|---|
| • t1, t2<br>• t1, t3, t2<br>• t1, t3, t4<br>• t1, t4 | • k1 | • v1, v2 |

**TABLE 15. Cas de test pour chaque machine en isolation**

### Etape 2: Le processus de marquage.

Supposons que le système à tester est modélisé par $C_1$, $C_2$,..., $C_k$ et supposons que nous voulons tester la machine $C_n$ dans le contexte.

Nous allons utiliser les cas de test générés par *EFTG* lors de l'étape 1 pour marquer toutes les transitions dans les autres machines qui vont participer à la génération du produit partiel. En d'autres mots, nous voulons marquer les transitions dans tous les chemins qui déclenchent $C_n$ (ou bien déclenchés par $C_n$), i.e., les cas de test des machines autres que $C_n$ qui contiennent des transitions envoyant des messages à $C_n$ (ou recevant des messages de $C_n$). Appelons le premier ensemble de transitions $Pr(C_n)$ et le second $Po(C_n)$. La génération des ensembles $Pr(C_n)$ et $Po(C_n)$ peut coûter très cher si l'analyse d'accessibilité exhaustive est utilisée et le problème qui se pose consiste à trouver les deux ensembles avec un effort minimal. Pour ce faire, notre méthode utilise les cas de test générés par l'outil *EFTG* lors de l'étape 1 comme guide. Si une transition dans $C_n$ reçoit (respectivement envoit) un message de (à) une *CEFSM* $C_i$ et puisque ce message est nécessairement envoyé (reçu) par au moins une transition de $C_i$ qui appartient à un ou plusieurs cas de test générés par *EFTG* pour $C_i$ en isolation, nous marquons toutes les transitions de ces cas de test. En marquant toutes les transitions dans chaque cas de test, nous nous assurons que les transitions précédant (resp. suivant) la transition qui envoit (resp. reçoit) le message participent à la génération du produit partiel. Après que le cas de test qui contient la transition envoyant (recevant) le message soit marqué, nous vérifions s'il contient des transitions recevant (envoyant) des messages (à) d'autres machines. Si tel est le cas, pour chacune de ces transitions T, la même procédure est répétée dans le but de marquer tous les chemins envoyant (recevant) le message reçu (envoyé) par T.

Le prochain algorithme marque toutes les transition de $C_n$, ensuite marque toutes les transitions des autres machines qui font partie des préambules et postambules des transitions de $C_n$. S'il existe des cycles entre 2 machines, l'algorithme Marquage les détecte, ce qui permet aux deux procédure récursives MarquageAvant et Marquage Arrière de terminer.

***Algorithme Marquage***

**Entrée**: La machine sous test $C_n$

**Sortie**: Marquage des transitions

**Début**

**Pour** chaque transition T de $C_n$ **Faire**

Marquer T;

**Si** T reçoit une entrée interne M d'une autre machine **Alors**

MarquageArriere (Emetteur de M, $C_n$, M)

**FinSi**

**Pour** chaque sortie interne O de T **Faire**

MarquageAvant($C_n$, Receveur de O, O)

**FinPour**

**FinPour**

**Fin**.

A ce stade, nous supposons que les cas de test pour chaque *CEFSM* (en isolation) sont déjà générés par *EFTG*. Le but de la procédure MarquageArriere (Emetteur, Receveur, M) est de marquer tous les chemins dans Emetteur qui contiennent une transition envoyant M à Receveur (i.e., de déterminer l'ensemble $Pr(C_n)$).

**Procédure MarquageArriere**

**Entrée**: Les processus Emetteur et Receveur du message M ainsi que le message M

**Sortie**: Transitions marquées

Début

**Pour** chaque cas de test TC de Emetteur **Faire**

**Si** il y a au moins une transition dans TC qui envoit M à Receveur **Alors**

**Pour** chaque transition **non-marquée** T dans TC **Faire**

**Si** T reçoit un message interne M' et si la procédure MarquageArriere n'a pas été appelée avec (Emetteur de M', Emetteur, M') **Alors**

MarquageArriere(Emetteur de M', Emetteur, M')

**FinSi**

**FinPour**

Marquer toutes les transitions dans TC

**FinSi**

**FinPour**

**Fin**.

De la même manière, la procédure MarquageAvant(Emetteur, Receveur, M) marque tous les chemins dans Receveur qui ont des transitions recevant le message M de Emetteur (i.e., déterminer l'ensemble $Po(C_n)$).

**Procédure MarquageAvant**

**Entrée**: Les processus Emetteur et Receveur du message M ainsi que le message M

**Sortie**: Transitions marquées

Début

**Pour** chaque cas de test TC de Emetteur **Faire**

**Si** il y a au moins une transition dans TC qui reçoit M de Emetteur **Alors**

**Pour** chaque transition **non-marquée** T dans TC **Faire**

**Pour** chaque instruction de sortie O de T telle que la procédure MarquageAvant n'a pas été appelée avec (Receveur, Receveur de O, O) **Alors**

MarquageAvant(Receveur, Receveur de O, O)

**FinPour**

**FinPour**

Marquer toutes les transitions dans TC

**FinSi**

**FinPour**

**Fin**.

Après cette étape, toutes les transitions marquées vont participer à la génération du produit partiel. Dans certains cas, l'usager peut vouloir tester une certaine *CEFSM* $C_n$ quand son contexte est un sous-ensemble des autres *CEFSM*s. Si c'est le cas, cette étape consiste à marquer toutes les transitions dans ce sous-ensemble qui communiquent avec $C_n$ directement ou indirectement.

Nous voulons générer le produit partiel de B3. Dans ce cas, toutes les transitions de B3 sont marquées. Puisque v1 envoie un message à B1, qui peut être reçu par t2, alors toutes les transitions de tous les cas de test de B1 qui contiennent t2 sont marquées (i.e., t1, t2 et t3).

Avant de marquer ces transitions, regardons d'abord les cas de test contenant t2. t1 reçoit un message externe (qui vient de l'environnement) et ne reçoit ni envoie aucun message interne. t2 envoie Int2 à B3, mais puisque toutes les transitions de B3 sont déjà marquées, rien ne se passe. t3 ne reçoit ni n'envoie aucun message interne. La transition k1 de B2 n'est pas marquée du fait qu'elle n'est influencée par aucune transition de B3, aussi, elle n'a aucune influence sur B1.

**Etape 3: Génération du produit partiel.**

Définition 6.2: Un produit partiel pour $C_n$ est défini par:

$PP(C_n) = C_n$ x {la machine formée par les transitions marquées}. $PP(C_n)$ est aussi appelé $C_n$ dans le contexte.

Après le processus de marquage, le produit partiel de $C_n$ est réalisé. A chaque instant, parmi les transitions possibles qui peuvent être choisies, seules les transitions marquées le sont. Les transitions non marquées ne participent pas au calcul du produit partiel car elles n'influencent pas la machine sous test. La procédure CalculPP calcule le produit partiel pour une certaine *CEFSM* et est similaire à un algorithme ordinaire d'analyse d'accessibilité. La différence majeure est que parmi les transitions possibles qui peuvent être choisies, seules les transitions marquées sont considérées.

Les fonctions Empiler (resp. Dépiler) sont des fonctions qui ajoutent (enlèvent) un état global à (de) la pile. Les procédures AjouterEtatGlobal (resp. AjouterTransitionGlobale) ajoutent un état global (resp. une transition globale) au graphe du produit partiel. La construction du graphe du produit partiel se fait donc par une exploration en **profondeur**. Une fois qu'un nouvel état est empilé, toutes les transitions marquées et possibles à partir de cet état sont ajoutées au graphe.

Tout d'abord, l'etat initial global (formé des états initiaux de chaque *CEFSM*) est créé et empilé. Par la suite, une transition possible dans une *CEFSM* C à partir de l'état global se trouvant en haut de la pile est exécutée. Si cette dernière doit consommer une donnée d'entrée, on vérifie si la donnée en entrée se trouve à la tête de sa file d'attente de C. Si tel est le cas, la donnée d'entrée est enlevée de la file d'attente. Sinon, une autre transition possible est choisie. Cette transition peut être une transition spontanée, ne nécessitant pas une donnée d'entrée, ou bien une transition qui doit consommer le message se trouvant à la tête de la file d'attente de C. Lorsqu'une transition est choisie, un nouvel état global est construit et est inséré dans le graphe d'accessibilité, s'il n'y existe pas. Aussi la nouvelle transition globale est ajoutée au graphe et une nouvelle exploration a lieu à partir du nouvel état global. Lorsque toutes les transitions possibles à partir d'un état global ont été traitées, ce dernier est enlevé de la pile et le traitement continue avec le nouvel état global, i.e., celui qui se trouve à la tête de la pile.

**Procédure CalculPP**
Entrée: la machine sous test M
Sortie: le produit partiel pour M
**Début**
Créer un nouvel état global NouvelEtat
AjouterEtatGlobal(NouvelEtat)
Empiler(NouvelEtat,Pile)
**Tant que** Pile non vide **Faire**
$EtatCourant \leftarrow Tete(Pile)$
**Si** une transition marquée T d'une certaine machine $C_i$ peut être exécutée **Alors**
Créer un nouvel état global NouvelEtat

**Si** la transition T possède une entrée interne **Alors**

Enlever la donnée d'entrée de la FIFO de $C_i$

**FinSi**

**Si** T possède des sorties internes **Alors**

Ajouter les données de sorties aux FIFOs correspondantes

**FinSi**

**Si** NouvelEtat n'existe pas dans le graphe d'accessibilité **Alors**

Ajouter EtatGlobal(NouvelEtat)

**FinSi**

AjouterTransitionGlobale(T)

Empiler(NouvelEtat)

**Sinon**

Depiler(Pile)

**FinSi**

**FinTQ**

**Fin**.

Après le processus de marquage, *CEFTG* calcule le produit partiel de B3. Ce dernier est montré dans la figure 6.3. Il contient 11 transitions et 6 états. Pour des raisons de clarté, nous avons gardé les anciens noms des transitions. Le détail sur chaque transition peut être trouvé dans la figure 6.2.

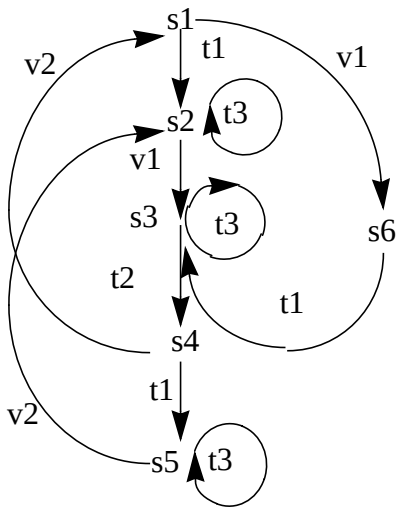



Figure 6.3 Le produit partiel de B3

**Etape 4: La génération de cas de test pour le produit partiel.**

Après que le produit partiel pour $C_n$ ait été calculé, ses cas de test sont générés automatiquement avec *EFTG* puisque le produit partiel est aussi une *EFSM*. Ces cas de test couvrent le flux de contrôle et de données dans le produit partiel et nous garantissons la couverture des critères “tous les def-use+ toutes les transitions” dans le produit partiel. Signalons aussi que notre méthode est plus adaptée pour tester une *CEFSM* dans le contexte quand le système global est formé de plus de deux *CEFSM*s car dans le cas contraire, le produit partiel pour une *CEFSM* n’est rien d’autre que le produit complet des deux *CEFSM*s. Le processus de génération des cas de test pour une *CEFSM* est résumé dans la figure 6.2.

Après la génération du produit partiel de B3, ses cas de test sont générés par *EFTG*. Ces cas de test couvrent tous les chemins “def-use” du produit partiel de B3. En termes de couverture de transitions, ils couvrent toutes les transitions de B3, 75% de transitions de B1 et aucune transition de B2. Ces cas de test sont représentés dans la table suivante:

| **Numéro** | **Cas de test** | **Numéro** | **Cas de test** |
|---|---|---|---|
| 1) | t1, t3, v1, t3, t2, v2 | 6) | t1, v1, t2, t1, v2, v1, t3, t2, v2 |
| 2) | t1, t3, v1, t2, v2 | 7) | t1, v1, t2, t1, v2, v1, t2, v2 |
| 3) | t1, v1, t3, t2, v2 | 8) | t1, v1, t2, v2 |
| 4) | t1, v1, t2, t1, t3, v2, v1, t2, v2 | 9) | v1, t1, t3, t2, v2 |
| 5) | t1, v1, t2, t1, v2, t3, v1, t2, v2 | 10) | v1, t1, t2, v2 |

**TABLE 16. Cas de test pour le produit partiel de B3**

Ces cas de test sont générés quand la valeur du paramètre en entrée *c* n'est pas connue. Puisque la valeur de *c* influence le flux de contrôle (du fait que le traitement à effectuer varie selon que *c* soit pair ou impair et *c* influence R), l'usager peut lire une valeur pour *c*.

Les prochains cas de test représentent les cas de test exécutables générés par *EFTG* quand la valeur de *c* est égale à 4 (dans toutes les transitions qui reçoivent Ext1(*c*)). Tous les cas de test ci-haut contenant t3 deviennent non-exécutables, puisque R ne peut jamais être impair. Cependant, les autres cas de test sont générés si on donne à *c* une valeur impair telle que 5 ou 9.

- Cas de test no 1: t1, v1, t2, t1, v2, v1, t2, v2
- Cas de test no 2: t1, v1, t2, v2
- Cas de test no 3: v1, t1, t2, v2.

De la même manière, pour calculer le produit partiel de B2, t1, t3, t4 et k1 sont marquées. Nous présentons ci-dessous les cas de test du produit partiel de B2. Ce dernier possède 7 transitions et 4 état.

| | | | |
|---|---|---|---|
| 1) | t1, t3, t4, k1 | 4) | t1, t4, t1, k1, t3, t4, k1 |
| 2) | t1, t4, k1 | 5) | t1, t4, t1, k1, t4, k1 |
| 3) | t1, t4, t1, t3, k1, t3, t4, k1 | 6) | t1, t4, t1, t3, k1, t4, k1 |

**TABLE 17. Cas de test pour le produit partiel de B2**

en termes de couverture de transitions, ces cas de test couvrent 75% des transitions de B1, 100% des transitions de B2 et 0% des transitions de B3. Si nous considérons les deux produits partiels, nous trouvons que leurs cas de test couvrent 100% des transitions de chaque machine ainsi que tous les chemins “def-use” pour les produits partiels de B2 et de B3.

La figure suivante résume le processus que nous venons de présenter.

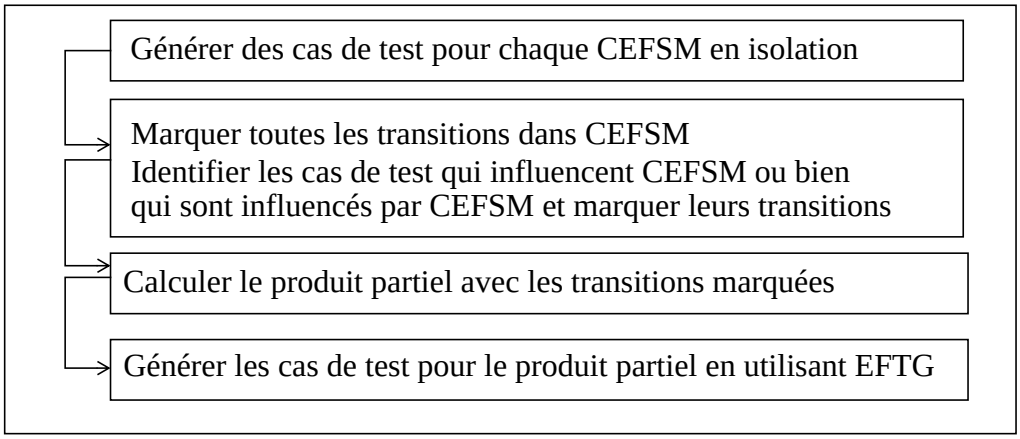



Figure 6.4 Le processus de génération de cas de test pour une CEFSM dans le contexte

Dans la prochaine section, le processus consistant à tester un système communicant est présenté.

## 6.4 Un algorithme de génération de cas de test pour les protocoles spécifiés par des CEFSMs

Dans cette section, nous présentons l’algorithme général *CEFTG* de génération automatique de cas de test pour les systèmes modélisés par des *CEFSM*s. Cet algorithme utilise l’outil *EFTG* ainsi que l’algorithme présenté dans la section 6.3. *CEFTG* génère des cas de test exécutables d’une façon incrémentale en générant des cas de test pour le produit partiel pour chaque *CEFSM* jusqu’à ce que la couverture désirée soit atteinte. Le processus de génération de cas de test pour le système global peut s’arrêter après que la génération du produit partiel pour certaines *CEFSM*s. Dans ce cas, ces cas de test couvrent tous les

chemins def-use pour les produits partiels (ceci inclut toutes les transitions dans la *CEFSM* pour laquelle le produit partiel a été calculé plus toutes les transitions des autres *CEFSMs*). En termes de couverture de transitions, ces cas de test couvrent 100% des transitions pour chaque *CEFSM* pour laquelle un produit partiel a été calculé et p% des transitions des autres machines, où p = (transitions marquées/ toutes les transitions).

Tout d'abord regardons l'exemple de la figure 6.5. Ce dernier est inspiré d'un système communicant existant. Le système global est formé de deux blocs A et B. Bloc A communique avec Bloc B et avec l'environnement, alors que Bloc B communique seulement avec Bloc A.

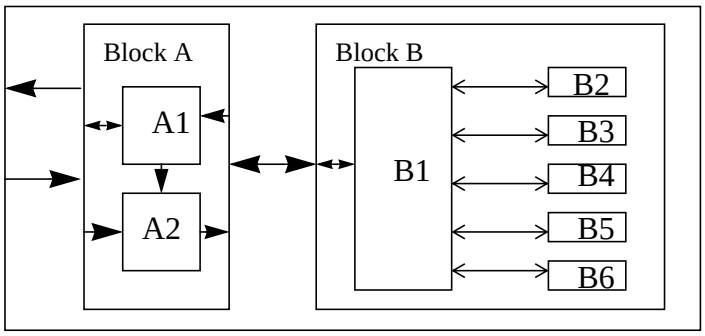



Figure 6.5 Architecture d'un système communicant

Dans ce cas particulier, B1 contrôle tous les autres processus du bloc B. Aussi aucun processus du bloc Bi, $2 \le i \le 6$, n'est influencé par un autre, i.e., aucun des messages envoyés par un processus n'a une influence sur les autres processus.

Supposons que nous voulons calculer le produit partiel de B1 quand son contexte est B; dans ce cas, toutes les transitions de B sont marquées et participent à l'analyse d'accessibilité réduite. En d'autres mots, le produit partiel de B1 est équivalent au produit total $B1 \times B2 \times B3 \times B4 \times B5 \times B6$. Ceci est dû au fait que B1 communique avec tous les autres processus du bloc B. Afin de tester le Bloc B, si au lieu de commencer avec le bloc B1, nous générons le produit partiel pour B2,..,B6, nous pourrons éviter de générer le produit partiel pour B1 (qui est le produit total de tous les processus du bloc B), car après la génération de ces produits partiels, les cas de test générés couvrent déjà toutes les

transitions de tous les processus (même celles de B1). Dans ce cas, si notre but est de simplement couvrir toutes les transitions de tous les processus, l'ordre suivi pour calculer les produits partiels s'avère très important.

Dans la prochaine section, nous présentons quelques mesures qui vont aider à déterminer l'ordre dans lequel les produits partiels sont calculés en commençant par la *CEFSM* qui a le plus petit nombre d'interactions avec les autres *CEFSM*s.

### 6.4.1 Mesures pour calculer la complexité de la communication

Dans la section 6.4.2, nous présentons un algorithme qui construit progressivement les cas de test pour un système communicant en générant des cas de test pour le produit partiel pour chaque *CEFSM*. Normalement, un produit partiel est généré pour chaque *CEFSM*. Mais dans certains cas, nous n'avons pas besoin de faire cela, du fait que les cas de test pour les produits partiels de quelques machines peuvent couvrir toutes les transitions de toutes les *CEFSM*s. Nous parlons ici de couverture de transitions et non de couverture de chemins def-use.

Pour des systèmes ayant une architecture similaire à celle du bloc B de la figure 6.5, il est plus intéressant de ne pas générer un produit partiel pour B1 et une méthode doit être utilisée pour éviter de calculer un produit partiel pour le bloc principal, qui dans ce cas est le produit complet de toutes les machines.

Soit M l'ensemble de toutes les *CEFSM*s. Nous définissons les mesures suivantes pour chaque *CEFSM* C.

- Compteur d'Entrées/Sorties (*CES*): Le nombre de messages internes en entrée et en sortie de C est considéré comme une mesure de la complexité de communication qui peut avoir lieu durant la composition des *CEFSM*s de M. *CES* de C est défini comme: *CES*(C) = le nombre de messages internes en entrée et en sortie mettant en cause les *CEFSM*s de M.

- Compteur de *CEFSM*s (*CC*): Le *CC* de C est le nombre de *CEFSM*s qui communiquent avec C.

Afin d'éviter la génération du produit complet lorsque le but est de couvrir tout simplement chaque transition de chaque *CEFSM*, nous devons d'abord calculer le *CES* pour chaque ***CEFSM***. A chaque itération, l'algorithme va calculer le produit partiel de la machine avec le plus petit *CES* et éventuellement avec le plus petit *CC*. Après le calcul du produit partiel, ses cas de test sont générés et l'usager peut arrêter l'exécution de l'algorithme quand la couverture escomptée est atteinte. Dans ce cas, pour le bloc B, l'exécution de l'algorithme peut arrêter lorsque les produits partiels pour B2, B3,.., B6 sont calculés, puisque, à ce moment, toutes les transitions de B1 sont déjà couvertes par les cas de test générés pour les produits partiels de B2,..,B6. Ceci est particulièrement intéressant si B1 est énorme puisqu'il communique avec toutes les autres machines du bloc B. Aussi, l'architecture du système présenté dans la figure 6.3 est très commune, et la méthode que nous présentons dans ce chapitre peut être très prometteuse pour de tels systèmes.

Dans la prochaine section, nous présentons un algorithme de génération automatique de cas de test pour un système composé de plusieurs *CEFSM*s, qui communiquent entre elles, en utilisant les mesures présentées ci-haut.

### 6.4.2 Un algorithme incrémental de génération de cas de test pour un système communicant

Le prochain algorithme présente notre méthode de génération de cas de test pour les systèmes communicants. L'usager a deux choix:

- Laisser l'algorithme générer un produit partiel pour chaque *CEFSM* ainsi que ses cas de test, ou

- Arrêter l'exécution de l'algorithme quand la couverture atteinte par les cas de test pour quelques produits partiels est satisfaisante (couverture des chemins def-use ainsi que toutes les transitions pour chaque produit partiel généré et couverture de transitions seulement pour chaque *CEFSM* pour laquelle un produit partiel n'a pas été calculé).

***Algorithme CEFTG***

**Entrée**: Nb: le nombre de *CEFSMs*, toutes les *CEFSMs*

**Sortie**: Cas de test pour le produit partiel de chaque *CEFSM*

**Début**

**Pour** i = 1 à Nb **Faire**

Lire chaque spécification *CEFSM*

Générer les cas de test pour *CEFSM* en isolation en utilisant *EFTG*

**FinPour**

$i \leftarrow 1$

**Tant que** $i \leq Nb$ **Faire**

Choisir la $CEFSM_i$ avec le plus petit CES (et CC) n'ayant pas encore été choisie

Marquage($CEFSM_i$)

$A_i \leftarrow CalculPP(CEFSM_i)$

Générer les cas de test exécutables pour $A_i$ en utilisant *EFTG*

CalculCouverture

**Si** l'usager est satisfait avec la couverture de transitions atteinte **Alors** Exit

**Sinon**

$i \leftarrow i + 1$

**FinSi**

**FinTQ**

**Fin**.

Après le calcul du produit partiel pour une ou plusieurs machines, la procédure CalculCouverture détermine le nombre de transitions dans chaque machine couverte par les cas de test générés (en termes de couverture de transitions).

***Procédure CalculCouverture***

Entrée: toutes les *CEFSMs*, tous les cas de test générés

Sortie: couverture atteinte par les cas de test pour chaque *CEFSM*

**Début**

**Pour** chaque CEFSM $C_i$ **Faire**

$Count[i] \leftarrow 0$

**Pour** chaque transition T de $C_i$ **Faire**

**Si** T est couverte par un cas de test généré par CEFTG **Faire**

$Count[i] \leftarrow Count[i] + 1$

**FinPour**

Afficher a l'usager (Count[i]*100)/Number of transitions in $C_i$)

**FinPour**

**Fin**.

Après le calcul des produits partiels de toutes les machines et la génération de leurs cas de test, nous vérifions si certaines machines ne sont pas complètement couvertes. Si tel est le cas, ceci veut dire que certaines transitions de certaines machines peuvent ne pas être accessibles quand la communication entre les différentes *CEFSM*s est considérée. Ceci peut être dû à des erreurs de spécification ou bien à des problèmes d'observabilité et de contrôlabilité.

## 6.5 Une technique de réduction de la taille du graphe d'accessibilité

Pour certains systèmes, l'ordre dans lequel une certaine *CEFSM* C peut recevoir des messages internes d'autres *CEFSM*s n'est pas important. Pour d'autres, il se peut que lorsqu'une des *CEFSM*s est dans l'état s, le premier message de sa file d'attente (*FIFO*) ne peut être consommé par aucune transition possible à partir de cet état.

Dans *SDL*, si un processus se trouvant dans un état s reçoit une donne d'entrée I et si aucune des transitions ayant comme état de départ l'état s n'est possible, le message I est écarté. Ceci est utilisé pour éviter certaines situations d'interblocage.

Aussi, une autre solution à ce problème consiste à utiliser la construction *SAVE* de *SDL*, où une donnée d'entrée qui ne peut être consommée par aucune transition est sauvegardée dans la *FIFO* du processus et pourra être consommée plus tard par une autre transition. Dans ce dernier cas, la file d'attente du processus n'est plus considérée comme une *FIFO* et le processus peut choisir le premier message dans la file qui peut être consommé par une transition de l'état s. Ce message n'est pas nécessairement le premier message de la file.

Pour de tels systèmes, nous proposons une méthode qui réduit d'une façon significative la taille du produit partiel généré par l'algorithme *CEFTG* (plus précisément par la procédure CalculPP).

La méthode peut aussi être utilisée pour tester le système global. L'idée est la suivante: deux états globaux M = ($m_1$, $m_2$,.., $m_k$) et N = ($n_1$, $n_2$,.., $n_k$) sont considérés équivalents si et seulement si:

- $\forall i$ = 1..k, $m_i = n_i$
- Soit $F_{mi}$ le contenu de la file d'attente de la *CEFSM* $C_i$ quand celle-ci est à l'état $m_i$. Alors, $\forall i$ = 1..k, $F_{mi}$ est équivalente à $F_{ni}$ ($F_{mi}$ et $F_{ni}$ sont constitués des mêmes messages mais l'ordre de présentation des messages dans les deux files n'est pas nécessairement le même).

Cette définition réduit considérablement la taille du graphe d'accessibilité (du produit partiel) et diminue la complexité de l'algorithme d'analyse d'accessibilité (partielle ou totale).

Cette méthode peut être utilisée pour tester les systèmes *SDL* utilisant la construction *SAVE*. Elle peut aussi être utilisée pour les systèmes ayant des processus pouvant recevoir des messages en retard (pour une raison ou une autre). Ceci peut causer l'arrivée d'autres messages (possiblement des messages moins prioritaires) à destination avant eux. (i.e., systèmes datagrammes).

En comparaison avec l'analyse d'accessibilité traditionnelle, le nombre d'états globaux est réduit considérablement puisque toutes les permutations des contenus des *FIFO*s sont maintenant considérées équivalentes.

Cette méthode peut aussi être très utile puisque nous comptons lier notre outil à un ensemble d'outils tel que *SDT*. La méthode propose d'utiliser une file d'attente pour chaque *CEFSM*. Cette file ne sera pas toujours une *FIFO*. Durant l'analyse d'accessibilité (ou bien le calcul du produit partiel), un message pouvant être consommé par une certaine transition est simplement récupéré de la file même s'il n'en est pas le premier élément et la

transition correspondante est déclenchée.

## 6.6 Génération de cas de test pour le produit complet de B1, B2 et B3

Nous avons calculé le produit complet des 3 machines de la figure 6.2 (voir figure 6.6). Ce dernier possède 30 transitions, 12 états et 1483 chemins def-use exécutables. Ce nombre correspond au produit total de trois petites machines.

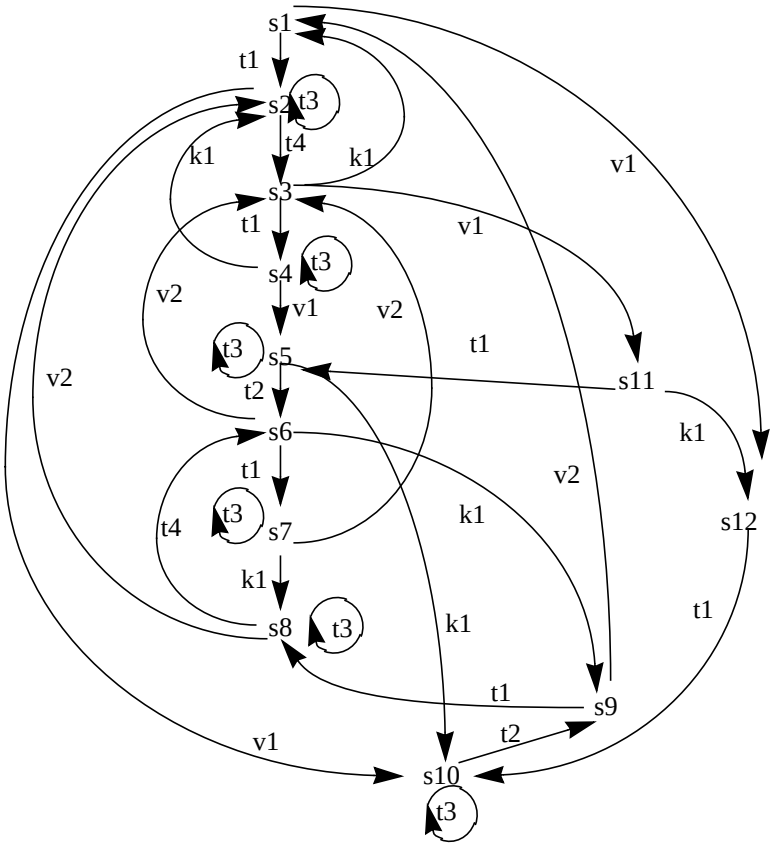



Figure 6.6 La machine correspondant au produit complet des *CEFSMs* de la figure 6.2

Pour des systèmes communicants composés de grandes *CEFSM*s, le calcul du produit complet devient infaisable. Pour cette raison, tester chaque *CEFSM* dans le contexte peut être la solution unique pour les systèmes larges.

## 6.7 Analyse des résultats

Tout d'abord, nous aimerions mentionner que tous les algorithmes présentés dans cette thèse ont été implantés en utilisant le langage C++ et ont été intégrés à *EFTG*. Dans l'exemple de la figure 6.2, nous pouvons clairement voir que k1 de B2 et que t4 de B1 n'ont aucune influence sur B3. Dans ce cas, l'algorithme de marquage a seulement marqué les transitions t1, t2, t3, v1 et v2.

Maintenant, comparons le produit partiel de B3 avec le produit complet de B1, B2 et B3. Si on enlève k1 du produit complet, les états s8, s9, s10 et s12 ainsi que leurs transitions sont aussi enlevés. Aussi, en enlevant t4 de la machine résultante, les états s2, s3 et leurs transitions sont enlevés. La nouvelle machine n'est rien d'autre que le produit partiel de B3 (voir figure 6.3).

En d'autres mots, le produit partiel de B3 est la projection du produit complet sur l'ensemble de transitions suivant: {t1, t2, t3, v1, v2}. Aussi, considérons les cas de test du produit partiel et les cas de test du produit complet. Si on enlève les cas de test contenant k1 et t4 alors 1475 cas de test sont enlevés en conséquence.

Afin de tester B3 dans le contexte, nous avons besoin de 10 cas de test seulement et non pas de 1483 cas de test. Les cas de test pour les produits partiels de B2 et de B3 peuvent être utilisés pour tester le système global, et nous n'aurons probablement pas besoin de générer le produit partiel de B1 qui est équivalent au produit complet de B1, B2 et B3.

Il est important de noter que les cas de test générés par notre méthode couvrent tous les états globaux ainsi que toutes les transitions globales de chaque produit partiel (mais pas les transitions globales du produit complet) puisque le calcul du produit partiel pour chaque *CEFSM* n'englobe pas toutes les transitions de toutes les *CEFSM*s. Ces cas de test couvrent aussi tous les chemins "def-use" de chaque produit partiel puisque la génération des cas de test est réalisée par *EFTG*.

Comparée aux autres méthodes telles que [Lee 96], notre critère de couverture est très fort. Dans ce cas, pour de gros systèmes, notre stratégie incrémentale de génération de cas de test peut être combinée avec une méthode d'analyse d'accessibilité réduite.

Dans la procédure CalculPP, à chaque étape, nous pouvons choisir une transition marquée qui n'a pas encore été testée pour réduire la taille du produit partiel. Bien entendu, le critère de couverture induit par cette méthode est plus faible. Aussi, l'utilisation d'une technique d'analyse d'accessibilité réduite telle que celle présentée dans la section 6.5 pour calculer le produit partiel réduit considérablement la complexité de la procédure CalculPP et permet de tester de plus gros systèmes.

Puisque nous divisons le problème de génération de cas de test pour un système communicant en générant d'une manière incrémentale des cas de test pour le produit partiel de chaque machine, nous divisons aussi la complexité du problème en ne considérant pas toutes les transitions de toutes les machines à la fois.

## 6.8 Complexité de CEFTG

Tout d'abord, nous allons regarder la complexité de la partie de l'algorithme qui calcule de produit partiel d'une *CEFSM* et qui génère des cas de test pour le produit partiel.

Soit S le système global composé de c *CEFSMs* et soit CT le nombre maximum de cas de test par machine.

Nous allons déterminer la complexité de chacune des étapes.

Complexité de l'étape 1

L'étape 1 consiste à générer des cas de test pour chaque *CEFSM* en isolation. Puisque ceci est réalisé à l'aide de l'outil *EFTG*, la complexité de cette étape est donc de $n^n$ si le critère "toutes les définitions-usages" est utilisé. Sinon, si le critère "tous les usages" est utilisé, la complexité serait de $n^6$.

Complexité de l'étape 2

Dans l'étape de marquage, les transitions de la machine sous test sont marquées. Si une de ces transitions reçoit ou envoie un message interne à une autre machine C, les cas de test de cette dernière sont examinés pour voir si eux aussi envoient ou reçoivent des messages internes à d'autres machines. Ceci est réalisé par les procédure Marquage-avant et Marquage-arrière. La complexité donc de la procédure de Marquage est égale à n * (Complexité (Marquage-avant) + Complexité (Marquage-arrière)), (n étant le nombre maximum de transitions, d'états ou de cas de test).

Calculons maintenant la complexité de Marquage-avant qui est la même que celle pour Marquage-arrière.

Tous les cas de test de la machine appelée par la procédure sont examinés pour voir si une de leurs transitions envoit ou reçoit un message interne d'une autre machine M. Si tel est le cas, les cas de test de M sont à leur tour examinés en appelant les procédures Marquage-avant et Marquage-arrière récursivement. Cependant, le nombre d'appels récursifs ne peut dépasser le nombre de machines. La complexité de cette étape est donc de l'ordre de $(CT * t)^c = k$ (pour chaque cas de test, toutes les transitions sont examinées et ce processus est répété au plus c fois (c étant le nombre de machines qui n'est généralement pas très grand)).

Par conséquent, la complexité de la procédure de Marquage est de n*k.

Complexité de l'étape 3

L'étape 3 est une sorte d'analyse d'accessibilité, et donc sa complexité est dans l'ordre de $t^t$ ou bien $n^n$.

Complexité de l'étape 4

L'étape 4 consiste à générer des cas de test pour le produit partiel généré à l'étape 3. Puisque cette génération est réalisée à l'aide *EFTG*, sa complexité dépend du critère de flux de données utilisé.

La complexité de la génération du produit partiel ainsi que de la génération de ses cas de test est donc de $n^n$. Ceci est dû au processus de génération du produit partiel qui est basé sur une technique d'analyse d'accessibilité.

## 6.9 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme *CEFTG* qui génère automatiquement et incrémentalement des cas de test exécutables pour un système communicant formé de plusieurs *CEFSM*s en générant des cas de test pour chacune des *CEFSM*s dans le contexte.

Il est important de souligner que cette méthode calcule un "produit partiel" pour chaque *CEFSM* avec des parties des autres *CEFSM*s. Aussi, puisque la méthode est guidée par les cas de test générés par *EFTG* (pour chaque *CEFSM* en isolation), et puisque dans le chapitre précédent, nous avons démontré sur quelques exemples que l'outil *EFTG* génère plus de cas de test que les méthodes existantes, nous sommes confiants que les produits partiels calculés par notre méthode sont représentatifs du comportement de chaque *CEFSM* dans son contexte.

Notre méthode a plusieurs avantages. Premièrement, cette méthode peut prendre moins de temps et d'espace que l'analyse d'accessibilité classique (impliquant toutes les transitions de toutes les machines). Deuxièmement, la génération de cas de test incrémentale peut réduire d'une manière significative l'effort pour re-tester un grand système à la suite d'une correction ou d'une amélioration.

Pour trouver les chemins contenant les transitions déclenchant (ou déclenchées par) la machine sous test, notre méthode utilise des informations générées par *EFTG* quand celui-ci est utilisé pour tester chaque machine en isolation.

Cette méthode peut aussi détecter certains interblocages dans une *CEFSM*, puisqu'elle utilise une technique d'analyse d'accessibilité et peut être utilisée pour détecter les transitions inaccessibles, en détectant les transitions qui ne sont pas couvertes par les cas de test générés.

Pour les systèmes *SDL* utilisant la construction “SAVE”, la technique présentée dans la section 6.5 peut être utilisée afin de rendre la taille du produit partiel raisonnable.

Comme mentionné plutôt, notre méthode peut facilement incorporer une méthode d’analyse d’accessibilité réduite telle que celle présentée dans [Lee 96] qui va réduire la taille du produit partiel de chaque *CEFSM*.

Finalement, puisque plusieurs méthodes publiées peuvent seulement être appliquées à de simples exemples ou bien à des systèmes communicants dont chaque composant est une *FSM*, nous considérons ce travail comme une étape vers le test du flux de données de vrais systèmes communicants.

# Chapitre 7

# Présentation de l'outil CEFTG[1]

## 7.1 Introduction

Dans ce chapitre, nous présentons l'outil que nous avons construit pour la génération automatique de cas de test et de séquences de test pour les systèmes modélisés par des *EFSM*s ou par des *CEFSM*s. Dans le chapitre 5, nous avons présenté l'outil *EFTG* qui génère des cas de test pour les systèmes modélisés par des *EFSM*s. Dans le chapitre 6, une extension de *EFTG* (*CEFTG*) qui génère des cas de test pour les systèmes modélisés par des *CEFSM*s a été présentée. *CEFTG* (**C**ommunicating **E**xtended **F**inite state machine **T**est **G**enerator) est un outil de génération de cas de test pour les protocoles de communication (ou les systèmes) modélisés par des machines à états finis communicantes (*CEFSM*s). Cet outil peut générer des cas de test pour des protocoles avec un seul processus en utilisant l'outil *EFTG*. Pour des systèmes avec plusieurs *CEFSM*s, l'usager a le choix de générer les cas de test pour tout le système global en effectuant une analyse d'accessibilité globale (en prenant en considération toutes les transitions de toutes les *CEFSM*s) et en générant les cas de test (en utilisant *EFTG*) pour le produit total. Il peut aussi générer les cas de test pour chaque *CEFSM* dans le contexte. Dans ce cas, le processus s'arrête quand la couverture atteinte par les cas de test générés est satisfaisante ou bien après avoir généré les cas de test pour tous les produits partiels (un produit partiel représentant le comportement d'une certaine *CEFSM* en considérant son interaction avec les autres machines et avec l'environnement).

1. Les résultats de ce chapitre sont publiés dans [Bour 99]

## 7.2 Notre méthodologie de génération de cas de tests à partir de systèmes SDL

Notre méthodologie utilise un ensemble d'outils développés à l'Université de Montréal. Certains de ces outils utilisent comme modèle sous-jacent le modèle de *FSM* et ont été adaptés afin d'utiliser le modèle *EFSM*.

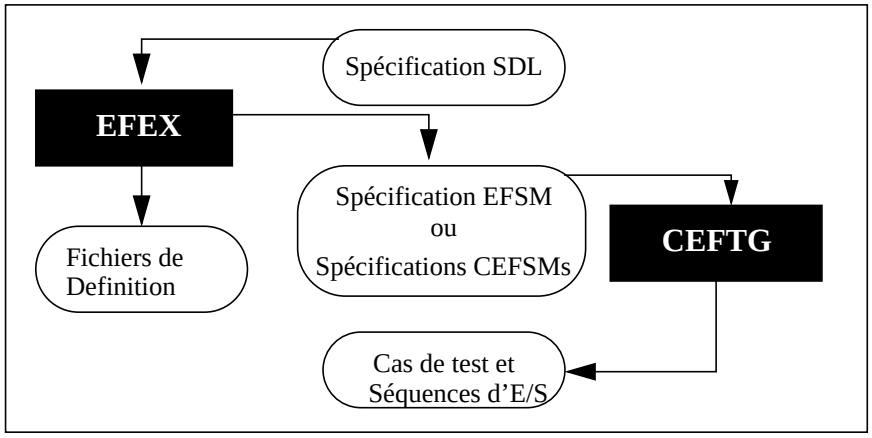



Figure 7.1 Processus de génération de cas de test à partir de spécifications SDL

La figure 7.1 montre le processus de génération de cas de test à partir de spécifications *SDL*. Tout d'abord, l'outil *EFEX* est appliqué sur le système *SDL* afin d'extraire l'*EFSM* correspondante si le système *SDL* est composé d'un seul processus. Dans le cas contraire, *EFEX* extrait une *CEFSM* pour chaque processus du système *SDL*. Parallèlement, des fichiers contenant les déclarations des signaux et des canaux sont crées. Pour ce qui est de l'outil *CEFTG*, il génère des cas de test exécutables pour les systèmes modélisés par des *CEFSMs*.

### 7.2.1 L'outil EFEX

*EFEX* est l'EFsm EXtractor. Il est basé sur l'outil *FEX* [Boch 97]. Ce dernier a été développé à l'Université de Montréal pour l'extraction d'une *FSM* représentant un comportement partiel d'un système *SDL*, en appliquant un algorithme de normalisation. Une ou plusieurs transitions dans la *FSM* générée correspondent à une donnée d'entrée déterminée à un état déterminé de la spécification *SDL*. Ceci est dû au fait que l'outil

utilise un dépliage partiel afin de préserver les contraintes sur les valeurs des paramètres en entrée. De plus, *FEX* génère d'autres fichiers qui peuvent être utilisés pour compléter les cas de test. Pour ces raisons, ainsi que pour la disponibilité de *FEX*, nous avons choisi de modifier *FEX* afin que le fichier généré par ce dernier puisse être traité par *CEFTG*. Malgré que *FEX* ait été construit à l'origine pour extraire une *FSM* à partir d'un processus *SDL*, il peut aussi être utilisé pour extraire l'*EFSM* correspondante. Dans [Boch 97], les fichiers générés par *FEX* contiennent des instructions d'affectation ainsi que des prédicats, mais du fait que ces informations ne sont pas utilisées par les *FSMs*, elles n'étaient pas utilisées par l'outil *TAG* décrit dans le même article, alors qu'elles vont être utilisées par *CEFTG*.

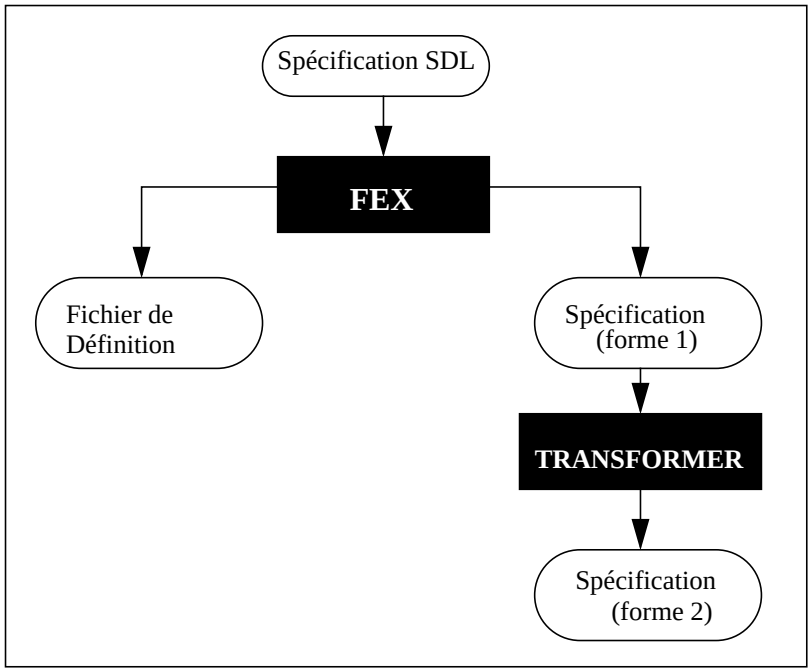



Figure 7.2 L'outil EFEX

Comme nous l'avons mentionné auparavant, l'entrée de *CEFTG* est une spécification sous forme normale, qui est différente de la sortie générée par *FEX*. Pour cette raison, le fichier généré par *FEX* (spécification (forme 1)) est traité par *TRANSFORMER* et transformé à la forme normale (Spécification (forme 2)).

La figure 7.3 ci-dessous, montre un exemple de spécification *SDL*, de la sortie de *FEX*

et de l'entrée acceptée par *CEFTG.*

```
Spécification SDL:
state S0;
    input I1;
    output O1;
    nextstate S1;

    input I2(n);
    decision n;
    (0): output O2;
        nextstate S2;
    (1): output O3;
        nextstate S3;
    enddecision;

Transitions générées par FEX:
S0?I1!O1 > S1;
S0?I2(n = 0)!O2 >S2;
S0?I2(n = 1)!O3 >S3;

Transitions générées par Transformer et acceptées par CEFTG:

When I1
From S0
To S1
Name t1: Begin
Output(O1);
End;

When I2(n)
From S0
To S2
Provided (n = 0)
Name t2: Begin
Output(O2);
End;

When I2(n)
From S0
To S3
Provided (n =1)
Name t3: Begin
Output (O3);
End;
```

Figure 7.3 Exemple de spécification *SDL* et de ses spécifications, forme 1 et forme 2, correspondantes

*EFEX* génère une *EFSM* (ou *CEFSM*) pour chaque processus d'un système *SDL*.

La prochaine section présente l'architecture de *CEFTG*. Ce dernier génère des cas de

test exécutables pour les systèmes modélisés par des *CEFSMs* en générant des cas de test pour chaque *CEFSM* dans le contexte.

### 7.2.2 CEFTG

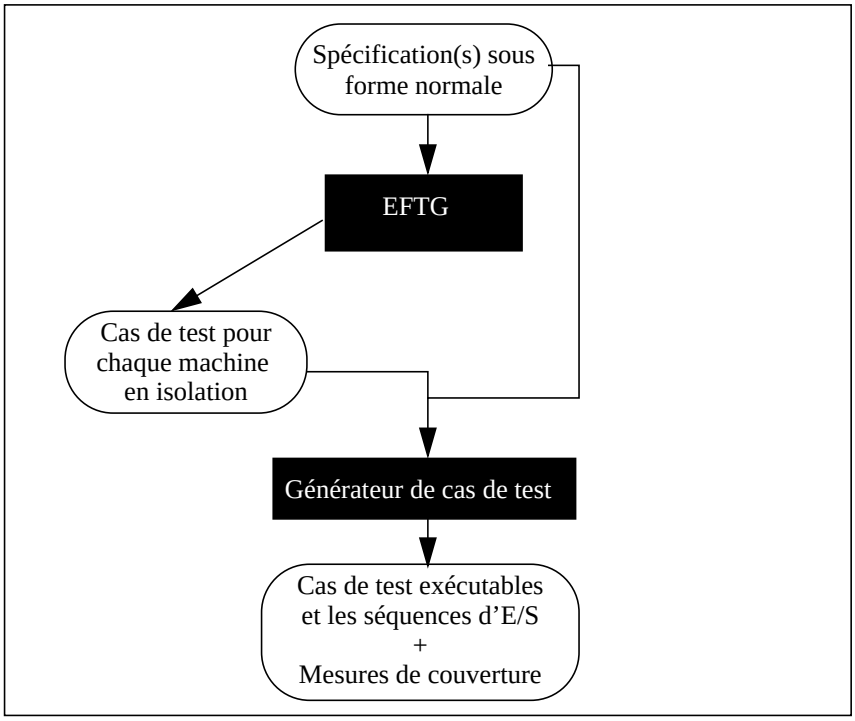



Figure 7.4 Architecture of CEFTG

*CEFTG* est le CEFsm Test Generator. Il peut générer des cas de test pour des systèmes modélisés par une *EFSM* en utilisant l'outil *EFTG*. Pour les systèmes modélisés par des *CEFSMs*, l'utilisateur peut générer des cas de test pour le système global en réalisant une analyse d'accessibilité complète, en prenant en compte toutes les transitions de toutes les *CEFSMs* et générer des cas de test pour le produit total. *CEFTG* peut aussi générer des cas de test pour chaque *CEFSM* dans le contexte. Dans ce cas, le processus termine quand la couverture atteinte par les cas de test générés est satisfaisante ou bien lorsque des cas de test sont générés pour tous les produits partiels.

*CEFTG* inclut toutes les activités en commençant par la spécification et en finissant par les cas de test et les séquences d'E/S. La première tâche réalisée par *CEFTG* est d'appliquer l'outil *EFTG* à chaque *CEFSM* afin de générer ses cas de test en isolation, i.e.,

quand la communication de cette *CEFSM* avec les autres n'est pas considérée. Durant cette phase, l'outil *EFTG* est appliqué afin de générer des cas de test pour chaque *CEFSM* en isolation. La figure 7.5 ci-dessous résume l'outil *EFTG*.

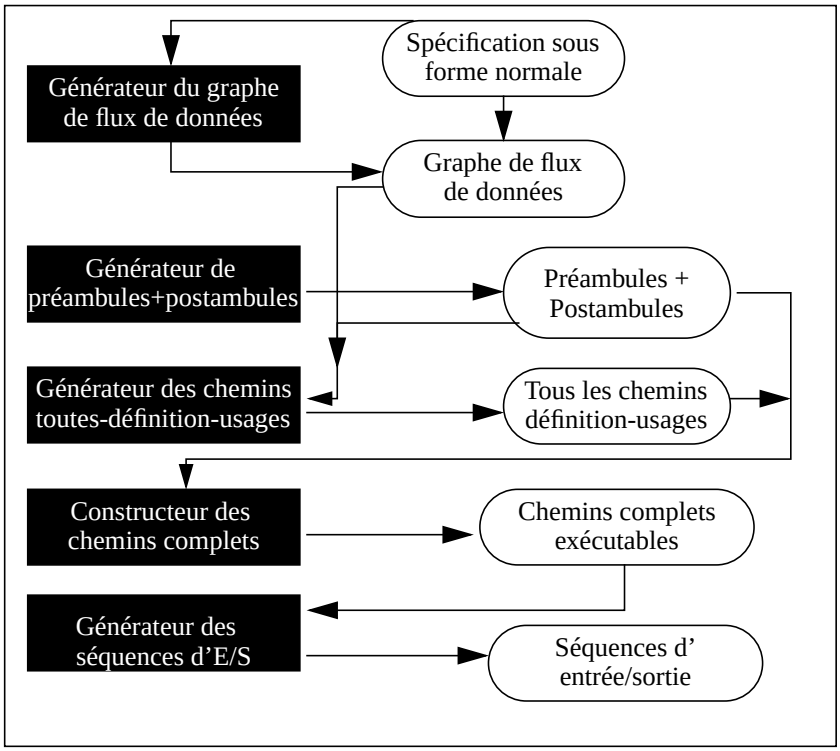



Figure 7.5 L'outil EFTG

**Le générateur de cas de test**

Le générateur de cas de test demande à l'utilisateur de faire un choix parmi les suivants:

- Générer des cas de test exécutables pour une certaine machine M dans son contexte: dans ce cas, il s'agit de marquer toutes les transitions de M et de voir quelles sont les autres machines qui communiquent avec M, i.e., qui envoient (resp. reçoivent) des messages internes à (resp. de) M. *CEFTG* marque donc tous les cas de test, i.e., toutes les transitions du cas de test, des autres machines envoyant (resp. recevant) des messages à (resp. de) M. Aussi, pour chacun des cas de test marqués, *CEFTG* vérifie si

une de ses transitions envoit (resp. reçoit) un message interne à (resp. d') une autre *CEFSM* (autre que M). Si tel est le cas, les cas de test de cette autre *CEFSM*, contenant des transitions recevant (resp. envoyant) des messages d' (resp. à) autres *CEFSMs* sont aussi marqués, et le processus de marquage continue jusqu'à ce que toutes les transitions participant dans la communication avec la machine M soient marquées. Une fois que le processus de marquage est fini, une analyse d'accessibilité réduite est effectuée en tenant compte des transitions marquées seulement. Cette analyse d'accessibilité va aboutir à un produit partiel qui va être utilisé par *EFTG* pour générer des cas de test. Ces derniers permettent donc de tester le comportement de la machine M quand son environnement est considéré. Son environnement est constitué de parties des autres *CEFSM*s qui communiquent avec M directement ou indirectement (via d'autres machines par une relation de transitivité).

- Générer des cas de test pour le produit total: dans ce cas, le processus de marquage n'a pas lieu du fait que toutes les transitions de toutes les machines participent à l'analyse d'accessibilité. *CEFTG* construit donc le produit total grâce à un algorithme d'analyse d'accessibilité classique, ensuite des cas de test sont générés pour le produit total. Cette option peut être utilisée si les machines sont de petite taille ou si l'usager est intéressé à tester le système global. Cependant, nous savons très bien que la génération de cas de test pour le produit total peut ne pas être possible à cause du problème de l'explosion combinatoire.

- Générer des cas de test pour chaque machine dans son contexte jusqu'à ce que les cas de test générés atteignent une certaine couverture ou bien que des cas de test soient générés pour chaque produit partiel: ce choix peut être pris lorsque l'usager s'intéresse seulement à générer des cas de test qui couvrent chaque transition d'une *CEFSM* et non pas chaque transition globale du système global (système formé par le produit de toutes les *CEFSMs*) ou bien s'il s'intéresse à couvrir une partie du système. Dans ce cas, *CEFTG* commence par générer le produit partiel, en commençant par la machine qui a le moins d'interaction avec les autres machines, le but étant d'éviter de générer de gros produits partiels. Chaque fois que des tests sont générés pour un produit partiel, un calcul de couverture est effectué pour déterminer combien de transitions sont

couvertes par les cas de test générés. Si l'usager est satisfait avec la couverture, *CEFTG* finit, sinon, *CEFTG* continue à calculer des produits partiels et à générer des cas de test pour ces derniers jusqu'à ce que l'usager soit satisfait ou bien lorsque tous les produits partiels sont calculés.

Une fois que les produits partiels sont générés, les séquences d'E/S sont extraites et sont utilisées pour tester la conformité de l'implémentation avec la spécification. Pour les messages d'entrée et de sortie avec paramètres, l'évaluation symbolique est utilisée afin de déterminer leur valeur.

## 7.3 Performances de CEFTG

Tout d'abord, rappelons que toutes les composantes de l'outil *CEFTG* ont été implantées à l'aide des langages C et C++.

Pour ce qui est de la rapidité, *CEFTG* donne des résultats très rapidement quand il s'agit de petits systèmes. Pour de gros systèmes, *CEFTG* peut prendre des heures, voire des jours avant de produire des résultats. Ceci est dû au critère "def-use" utilisé. L'utilisation d'un critère plus faible, tel que le critère "tous les usages", influencera énormément la rapidité du programme de génération de cas de test. Il faut aussi noter que le matériel utilisé pour tester *CEFTG* n'est pas des plus performants et un matériel plus rapide aurait probablement influencé le temps d'exécution.

Pour ce qui est de l'espace mémoire, *CEFTG* n'a pas pu générer des cas de test pour un gros système formé de plusieurs processus communicants quand le critère "def-use" a été utilisé mais l'a fait avec le critère "tous les usages".

Dans le cas de gros systèmes, la lenteur de *CEFTG* ainsi que la quantité de mémoire utilisée sont attribuées à sa forte complexité (l'algorithme qui génère tous les chemins def-use entre deux transitions T1 et T2 est très récursif).

L'utilisation du critère "def-use" peut ralentir d'une manière considérable la génération de cas de test. De plus, le nombre de cas de test générés pour un système à l'aide ce critère est généralement très grand. Pour toutes les raisons mentionnées

précédemment, il peut s'avérer beaucoup plus avantageux d'utiliser le critère "tous les usages" ou un critère plus faible pour générer des cas de test pour des systèmes réels.

Pour ce qui est des exemples présentés dans cette thèse, la génération de cas de test était très rapide (quelques secondes).

Regardons maintenant la figure 7.6. Cette dernière présente l'architecture d'un vrai système de télécommunication qu'on ne pourra pas détailler pour des raisons de confidentialité.

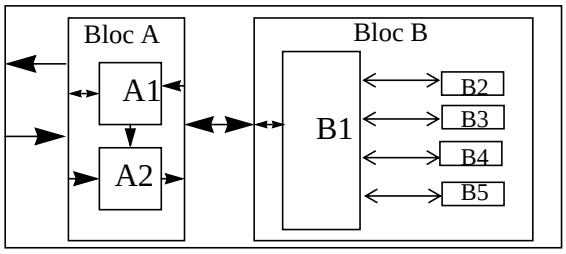



Figure 7. 6 Architecture d'un vrai système de télécommunication

Nous avons exécuté *CEFTG* sur le système de télécommunication composé de 6 *CEFSM*s (représenté par le bloc B); les résultats suivants ont été obtenus. PP dénote le produit partiel.

| CEFSM | nombre d'états | nombre de transitions | nombre d'états du PP | nombre de transitions du PP | nombre de cas de test du PP |
|---|---|---|---|---|---|
| 1 | 2 | 17 | - | - | - |
| 2 | 11 | 24 | 1027 | 3992 | - |
| 3 | 1 | 3 | 8 | 13 | 9 def-use |
| 4 | 4 | 10 | 24 | 55 | 43 def-use |
| 5 | 5 | 12 | 75 | 139 | 95 chemins tous-les-usages |
| 6 | 3 | 11 | 360 | 1262 | 2355 chemins tous-les-usages |

**TABLE 18. Résultats de l'application de CEFTG sur un vrai exemple**

Nous pouvons voir que dès que la taille du produit partiel devient importante, la génération de cas de test pour le produit partiel devient très lente, sinon impossible (dû à

un problème de saturation de mémoire). Ceci est dû au critère "toutes les définition-usages" utilisé. Aussi, la *CEFSM* 1 communique avec toutes les autres machines et son produit partiel serait équivalent au produit total (impliquant toutes les machines). Son produit partiel n'a pas pu être généré (problème d'explosion d'états dû à l'analyse d'accessibilité). Notons aussi que les *CEFSMs* de cet exemple comportent beaucoup de variables et de prédicats.

## 7.4 Conclusion

Nous avons présenté dans ce chapitre l'architecture de l'outil *CEFTG*. Ce dernier est assez simple à utiliser et ne requiert aucune connaissance spéciale de la part de l'utilisateur. L'outil guide la génération des cas de test en posant des questions simples à l'utilisateur afin de déterminer quel critère sera utilisé ("toutes les définitions-usages" ou "tous les usages" et pour savoir si des cas de test vont être générés pour un seul ou pour tous les produit partiels.

# Chapitre 8

# Conclusion

Dans cette thèse, nous avons proposé une méthodologie pour la génération de cas de test pour des protocoles de communication ainsi que pour tout système pouvant être modélisé par plusieurs *CEFSMs* qui communiquent les unes avec les autres via des files *FIFO*.

Tout d'abord, nous résumons cette thèse, ensuite, nous soulignons nos contributions majeures, nos réalisations ainsi que les extensions possibles à notre méthode.

## 8.1 Resumé

Etant donné que les logiciels de communications sont un cas particulier de logiciels, nous avons présenté, dans le deuxième chapitre, les différentes activités du cycle de développement d'un logiciel. Nous avons expliqué en détail la phase de spécification étant donné que c'est une des phases les plus importantes du cycle de développement de logiciel ainsi que l'avantage de l'utilisation des *TDF*s pour la spécification des systèmes.

Dans le troisième chapitre, nous avons parlé de l'activité de test, qui est une des activités du cycle de vie d'un logiciel. Nous définissons les différentes stratégies de test, les types de test, les architectures de test et la couverture de test.

Cependant, avant de tester un logiciel de communication (ou tout système), ce dernier doit d'abord être spécifié. Le quatrième chapitre présente les modèles utilisés pour la spécification des systèmes, notamment le modèle de *FSM* et ses extensions. Aussi, le

modèle des *réseaux de* Petri ainsi que la logique formelle qui constituent d'autres formalismes pour la spécification des systèmes, ont été présentés. Dans une deuxième partie de ce chapitre, nous avons ennoncé les méthodes existantes de génération de cas de test dédiées au test du flux de contrôle et du flux de données. Nous avons également présenté un panorama d'outils de génération de cas de test à partir de spécifications écrites dans le langage *SDL*. Notons que la plupart de ces outils sont semi-automatiques et nécessitent que les buts de test soient spécifiés par l'utilisateur.

Ainsi, après avoir fait le tour des méthodes existantes de génération de cas de test, nous présentons dans le cinquième chapitre notre méthode de génération de cas de test pour les systèmes modélisés par une seule *EFSM*. Cette méthode est une méthode unifiée pour le test du flux de contrôle et du flux de données. Elle génère des cas de test exécutables pour tout système modélisé par une *EFSM*. Aussi, elle génère un plus grand nombre de cas de test que les autres méthodes étant donné qu'elle utilise l'analyse de cycle qui consiste à insérer le cycle influent un certain nombre de fois dans un chemin non exécutable afin de le rendre exécutable. Elle permet également de générer des tests pour des systèmes contenant des boucles non bornées.

Généralement, les spécifications des protocoles de communication et des systèmes en général sont constituées de plus d'une *EFSM*. La méthode présentée dans le cinquième chapitre n'est donc plus suffisante. Pour cette raison, nous avons développé une méthodologie de génération de cas de test pour les systèmes modélisés par plusieurs *CEFSMs*, que nous présentons dans le sixième chapitre. Cette méthodologie teste un système en générant des cas de test pour chacune de ses *CEFSMs* dans le contexte, i.e., en considérant sa communication avec les autres *CEFSM*s. Au lieu de générer le graphe d'accessibilité pour le système global, ce dernier est testé d'une manière incrémentale afin d'éviter le problème de l'explosion combinatoire. Cette méthodologie est pratique et permet donc de tester les systèmes réels.

Le septième chapitre, quant à lui, présente l'architecture de l'outil *CEFTG* (Communicating Extended Finite State machine Generator) ainsi que ses principales composantes.

Le huitième et dernier chapitre résume tout ce qui a été présenté dans cette thèse et présente les extensions possibles de ce travail.

## 8.2 Contributions et réalisations

Notre première contribution concerne le développement d'une méthode de génération de test unifiée qui permet de générer des cas de test pour les systèmes modélisés par des *EFSMs* et l'outil *EFTG* en est la réalisation. Cette méthode utilise des techniques pour le test du flux de contrôle ainsi que pour le test du flux de données. Pour ce qui est du test du flux de contrôle, nous avons choisi d'utiliser les séquences *UIO* à cause de ses multiples avantages. Pour le flux de données, le critère toutes-les définition-usages ou "def-use" a été utilisé. Ce dernier n'est pas le critère le plus fort, mais permet de tester tous les chemins partant d'une définition d'une variable et finissant à tous ses usages. L'avantage majeur de notre approche par rapport aux autres méthodes est qu'elle génère des cas de test exécutables. Pour ce faire, l'analyse de cycles est utilisée pour trouver le cycle le plus court pouvant être inséré dans un chemin afin de le rendre exécutable. L'évaluation de l'exécutabilité d'un chemin est possible grâce à l'exécution symbolique. Etant donné que le critère "def-use" peut ne pas convenir pour les gros systèmes, nous avons également implanté le critère "tous les usages" qui consiste à générer un seul cas de test entre une définition d'une variable et un de ses usages afin de tester un plus grand nombre de systèmes. Notons aussi que *EFTG* peut être utilisé pour des systèmes contenant des boucles non bornées. Une fois que les chemins exécutables sont générés, ils sont complétés afin de former des cas de test complets, puis sont utilisés pour générer les séquences d'E/S qui vont servir à l'évaluation de la conformité de l'implantation avec la spécification.

L'autre contribution majeure concerne le test des systèmes modélisés par plusieurs machines communicantes ou *CEFSMs*. Nous avons développé une méthodologie pour le test de tels systèmes. Cette dernière teste le système en testant chacune de ses *CEFSMs* dans le contexte. Pour ce faire, l'outil *CEFTG*, réalisé, génère un produit partiel pour chaque *CEFSM*. Un produit partiel étant la machine composée de la *CEFSM* et des parties des autres CEFSMs qui communiquent avec elles directement ou indirectement (à travers

d'autres *CEFSMs*). Une fois le produit partiel calculé, ses cas de test sont générés à l'aide de l'outil *EFTG*. Ces cas de test couvrent donc le critère de flux de données de *EFTG*, notamment le critère def-use, i.e., tous les chemins def-use du produit partiel sont générés. Ces cas de test couvrent toute la machine initiale, i.e., celle pour laquelle le produit partiel est calculé ainsi que des parties (des transitions) d'autres *CEFSMs*. Ainsi, pour tester le système global, chaque *CEFSM* est testée dans le contexte, i.e., le produit partiel de chaque *CEFSM* est généré. Ce processus termine lorsque tous les produits partiels sont générés ou bien lorsque la couverture atteinte par les cas de test des produits partiels générés est satisfaisante. Cette méthodologie permet donc d'éviter de calculer le produit global, i.e., le graphe d'accessibilité, du système pour le tester. Ce dernier est testé d'une manière incrémentale afin d'éviter le problème de l'explosion combinatoire, rendant ainsi possible le test de systèmes réels.

Nous avons également proposé dans le cadre de cette thèse une méthode qui réduit la taille du produit partiel. Cette dernière peut aussi être utilisée pour réduire la taille du produit total. Cette méthode considère que des états globaux dont les files d'attente contiennent les mêmes messages sont équivalents même si l'ordre des messages dans les deux files n'est pas identique. Cette hypothèse réduit d'une manière considérable la taille du graphe d'accessibilité. Cette méthode peut être utilisée pour les systèmes où l'ordre d'arrivée des messages n'est pas important.

Finalement, parmi les outils que nous avons développé, l'outil Transformer permet à *CEFTG* de générer des cas de test pour les spécifications écrites dans le langage *SDL*.

## 8.3 Extensions possibles

Nous avons implanté l'outil *EFTG* en premier, puis nous lui avons intégré les algorithmes présentés dans le chapitre 7 qui concernent la génération de test pour les systèmes modélisés par des *CEFSMs*. Néanmoins, il reste plusieurs avenues à explorer et il serait intéressant d'implanter d'autres boîtes afin de compléter l'outil.

- Augmenter les sous-ensembles des constructions *SDL* permises afin de pouvoir tester un plus large ensemble de systèmes *SDL*.

- Transformer les cas de test générés par *CEFTG* en cas de test *TTCN* afin de pouvoir les utiliser pour tester toute implantation du système.

- Nous avons implanté l'algorithme Dijkstra [Dijk 59] afin de trouver les plus courts préambules et postambules pour chaque état. Comme le plus court préambule à un certain état peut ne pas être exécutable, il serait intéressant d'implanter d'autres algorithmes qui trouvent tous les plus courts chemins entre deux états ou bien les chemins de longueur k [Moni 85, Wata 81].

- Implanter les critères de flux de données présentés dans [Weyu 85] qui sont plus faibles que les critère "def-use" et "tous les usages" et qui permettront sûrement de tester certains aspects de systèmes communicants plus complexes.

- Implanter des techniques pour réduire la taille du graphe d'accessibilité telle que celle présentée dans [Lee 96] afin de pouvoir tester des systèmes complexes.

# Bibliographie

[Aho 88] Alfred V.Aho, Anton T. Dahbura, David Lee, M.Umit Uyar, “An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours”, Protocol Specification, Testing and Verification VIII, IFIP, 1988.

[Alga 95] Bernard Algayres, Yves Lejeune, Florence Hugonnet (Verilog), “GOAL: Observing SDL behaviors with ObjectGeode”, 7th SDL Forum, Oslo, Norway, 26-29 September 1995.

[Alur 94] Rajeev Alur, David Dill, “A Theory of Timed Automata”, Theoretical Computer Science, pp. 183-235, 1994.

[Andr 83] G.R. Andrew, F.B. Schneider, “Concepts and Notation for Concurrent Programming”, ACM Computing Surveys, 15(1):3-34, 1983.

[Andr 85] Stephen J. Andriole, “Software Validation Verification Testing and Documentation”, Petrocellt Books, 1985.

[Anid 95] Ricardo Anido, “Guaranteeing Full Fault Coverage for UIO-Based Testing Methods”, International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September, 1995.

[Anid 96] Ricardo Anido, Ana Cavalli Toma Macavei, Luiz Paula Lima, Marylene Clatin, Marc Phalippou, “Engendrer des tests pour un vrai protocole grâce à des techniques éprouvées de vérification”, CFIP’96, Rabat, Maroc, pp. 500-513, 1996.

[Bani 87] B.Banieqbal, H.Barringer, A.Pnueli, “Temporal Logic in Specification”, Proceedings, Springer-Verlag, 1987.

[Beiz 90] Boris Beizer, “Software Testing Techniques”, Van Nostrand Reinhold, New-York, 1990.

[Beli 89] Ferenc Belina, Dieter Hogrefe, “The CCITT-Specification and Description Language SDL”, Computer Networks and ISDN Systems, Vol. 16, 1989.

[Beli 91] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma, “SDL with Applications from Protocol Specification”, Prentice Hall International, 1991.

[Bert 91] Bernard Berthomieu, Michel Diaz, “Modeling and Verification of Time Dependent Systems Using Time Petri Nets”, IEEE Trans. on Software Engineering, vol. 17, No 3, March 1991.

[Bisw 86] Biswajit Kanungo, Louise Lamont, Robert L. Probert, Hasan Ural, “A Useful FSM Representation For Test Suite Design and Development”, Proc. Sixth Int’l Workshop Protocol Specification, Testing, and Verification, pp. 163-176, 1986.

[Bnr 1988] Bnr, “BNR Prolog Reference Manual”, Bell-Northern Research 1988.

[Bnr 1988] Bnr, “BNR Prolog User’s guide”, Bell-Northern Research 1988.

[Boch 78] Gregor V. Bochmann, “Finite State Description of Communication Protocols”, Computer Networks, Vol. 2, pp. 361-372, 1978.

[Boch 80b] G. v. Bochmann, “Specification and Verification of Computer Communication Protocols”, chap. 5 in “Advances in Data Communications Management”, ed. T.A. Rullo, Heyden Publ. 1980.

[Boch 90] G. v. Bochmann, “Protocol Specification for OSI”, Computer Networks and ISDN Systems, Vol. 18, pp. 167-184, 1989-90.

[Boch 91d] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi et G. Luo, “Fault Models in Testing”, Proc. IFIP Intern. Workshop on Protocol Test Systems, Netherlands, pp. (II-17)-(II-32), Octobre 1991.

[Boch 92] G. v. Bochmann an al. “Fault Models in Testing”, Dans R. J. H. J. Kroon and E. Brinksma, editors, IFIP Transactions, Protocol Test Systems, IV Proceeding of IFIP TC6 4th International Workshop on Protocol Test Systems, 1991), pp. 17-30, North-Holland.

[Boch 94a] G.v. Bochmann et A. Petrenko, Protocol Testing, “Review of Methods and Relevance for Software Testing”, publication #923, DIRO, Univérsité de Montréal, 1994.

[Boch 94b] G. v. Bochmann, A. Petrenko et M. Yao, “Fault Coverage of Tests based on Finite State Models”, (invited paper) IFIP Intl Workshop on Protocol Test Systems, Tokyo 1994.

[Boch 97] G. v. Bochmann, A. Petrenko, O. Belal et S. Maguiragua, “Automating the Process of Test Derivation from SDL Specification”, the Eighth SDL Forum 97, INT Evry France, 22-26 Septembre 1997.

[Bolo 87] T. Bolognesi et E. Brinksma, “Introduction to the ISO Specification Language LOTOS”, Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1987.

[Bour 99] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, “A Test Case Generation Tool for Conformance Testing of SDL Systems”, 9th SDL Forum, pp. 405-419.

[Bour 98] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, “A guided incremental test case generation method for testing CEFSM based systems”, IWTCS’98, Tomsk, Russie, Kluwer Academic Publishers, pp.279-294.

[Bour 97] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, “Automatic executable test case generation for EFSM specified protocols”, IWTCS, Chapman & Hall, pp. 75-90.

[Brin 87] Ed Brinksma, Giuseppe Scollo, Chris Steenbergen, “LOTOS Specifications, their Implementations and their Tests”, Proc. Sixth Intern.Workshop Protocol Specification, testing and Verification, pp. 349-360, 1987.

[Brin 93] E. Brinksma, “Sur la couverture des validations partielles”, Actes du colloque Francophone sur l’Ingénierie des Protocoles, CFIP’93, Montréal, 1993.

[Brom 89] Lars Bromstrup, Dieter Hogrefe, “TESDL: Experience with Generating Test Cases from SDL Specifications”, Fourth Proc. SDL Forum, pp. 267-279, 1989.

[Budd 83] T. A. Budd, “Techniques for advanced sofware validation”, Sofware Engineering: Developments State of the Art Report, 1983.

[Budk 86] T. Budkowski, P. Dembinski et J. P. Ansart, “Estelle, Un Langage de Spécification des Systèmes Distribués”, 3eme Congrès ‘De nouvelles Architectures pour les communications’ Paris 28-30 Octobre1986.

[Budk 87] T. Budkowski, P. Dembinski, “An Introduction to Estelle: a Specification Language for Distributed Systems”, Computer Networks and ISDN, Vol. 14, No. 1, 1987.

[Cacc 93] Léo Cacciari, Omar Rafiq, “La Validation Réduite des Protocoles de Communication”, Rapport technique du Laboratoire TASC, Univérsité de Pau, Septembre 1993.

[CCIT 92] CCITT, “SDL Methodology Guidelines”, Appendix I to Recommendation Z.100, June 1992.

[Chan 93] Samuel T. Chanson, Jinsong Zhu, “A Unified Approach to Protocol Test Sequence Generation”, in Proc. IEEE INFOCOM, San Francisco, March 1993.

[Chan 94] Samuel T. Chanson, Jinsong Zhu, “Automatic Protocol Test Suite Derivation”, IEEE 1994.

[Char 96] Olivier Charles, Roland Groz, “Formalisation d’Hypothèses pour l’Evaluation de la Couverture de Test”, CFIP’96, Rabat, Maroc, pp.483-497, Octobre 1996.

[Chow 78] T.S.Chow, “Testing Software Design Modelled by Finite State Machine”, IEEE Trans on Software Engineering, 4, No 3, May 1978.

[Chun 90] Woojok Chun, Paul D. Amer, “Test Case Generation for Protocols Specified in Estelle”, FORTE’ 90, Madrid, Nov 1990.

[Clar 76] L. Clarke, “A System to Generate Test data and Symbolically Execute Programs”, IEEE Transactions on Software Engineering, 2(3), Septembre 1976.

[Clar 85] Lori A. Clarke, Debra J. Richardson, “Applications of Symbolic Evaluation”, The Journal of Systems and Software 5, pp 15-35, 1985.

[Clar 86] E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications”, ACM Transactions on Programming Languages and Systems, 8(2):244-263.

[Clat 95] Marylene Clatin, Roland Groz, Marc Phalippou, Richard Thummel, “Two Approaches Linking a Test Generation Tool with Verification Techniques”, International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September, 1995.

[Come 97] E. R. Comer “Alternative Software Life Cycle Models”, Software Engineering, Merlin Dorfman & Richard H. Thayer Eds, Forwarded by Barry W. Bohem, 1997.

[Day 83] J.D. Day, H. Zimmermann, “The OSI Reference Model”, Proc. IEEE, Vol. 71, No. 12, pp. 1334-1340.

[Dech 89] R. Dechter, J. Pearl, “Tree Clustering for Constraint Networks”, Artificial Intelligence, 38:353-366, 1989.

[DeMi 91] Richard A. DeMillo, Jefferson Offutt, “Constraint-Based Automatic Test Data Generation”, IEEE Transactions on Software Engineering, Vol. 17, No. 9, September 1991.

[Dijk 59] E. W. Dijkstra, “A note on two problems in connection with graphs”, Numer. Math. 1 (1959) pp. 269-271.

[Drus 89], Doron Drusinsky, David Harel, “Using Statecharts for Hardware Description and Synthesis”, in IEEE Transactions on Computer-Aided Design, 1989.

[Dsso 99] R. Dssouli, K.Saleh, E.M. Aboulham, A. Ennouaary, C. Bourhfir, “Test Development for Communication Protocols: Towards Automation”, Special Issue “Advanced Topics on SDL and MSC”, Computer Networks, pp. 1835-1872, 7 Juin 1999.

[Dubu 92] M. Dubuc, R. Dssouli et G. v. Bochmann, “TESTL: A Tool for the Analysis of Test Sequences based on Finite-State Model”, IWPTS'92, IFIP Transactions, Protocol Test Systems IV, North Holland Publ. pp.195-206, 1992.

[Ehri 89] H. Ehrig, B. Mahr, “Fundamentals of Algebraic Specification 1, EATCS Monographs on Theoretical Computer Science”, 6, Springer-Verlag, Berlin, 1989.

[ElMa 93] K. El Maadani, “Identification de Systèmes Séquentiels Structurés. Application à la Validation du Test”, Thèse INSA Toulouse, 1993.

[Ells 92] J. Ellsberger et F. Kristoffersen, “Testability in the context of SDL”, In R. J. Linn and M. U. Uyar, editors, Protocol Specification Testing and Verification XII, Amsterdam, North-Holland, 1992.

[Este87] ISO DIS9074, “Estelle: A Formal Description Technique Based on Extended State Machine Model”, 1987.

[Fern 96] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, Laurence Nedelka, Cesar Viho, “An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology”, Publication Interne IRISA 1035, Juin 1996.

[Fike 70] R. E. Fikes, “A System for Solving Problems Stated as Procedures”, Artificial Intelligence, 1, pp. 27-120, 1970.

[Fuji 90] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou et A. Ghedamsi, “Test Selection Based on Finite State Models, Publication départementale #716, DIRO, Université de Montréal, Février 1990.

[Fuji 91] Susumu Fujiwara, Gregor V. Bochmann, Ferhat Khendek, Mokhtar Amalou, Abderrazak Ghedamsi, “Test Selection Based on Finite State Models”, IEEE Trans.

Software Eng., Vol.17, No. 6, pp. 591-603, June 1991.

[Gadr 90] J.Gadre, C. Rohrer, C. Symington “A COS Study of OIS Interoperability”, Computer Standards & Interfaces, 9, 217-237, 1990.

[Gaud 96] M. C. Gaudel, B. Marre, F. Schlienger, G. Bernot “Précis de génie logiciel”, Masson, 1, paris, 997.

[Genr 87] H. Genrich, “Predicate/Transitions Nets: Central Models and their Properties”, Lecture Notes in Computer Science 254, pp. 207-247, (W. Bauer, W. Reisig, R. Rozenberg, editors) Springer: 1987.

[Gidd 84] R. V. Giddings “Accomodating Uncertainty in Software Design”, Comm. ACM, Vol. 27, No. 5, pp. 428-434, 1984.

[Gill 1962] A. Gill, “Introduction to the Theory of Finite State Machines”, McGraw Hill, 1962.

[Gode 93] Patrice Godefroid, Gerard Holzmann, “On the verification of Temporal Properties”, In the 13th International IFIP WG 6.1 Symposium on Protocol Specification, Verification and Testing.

[Gone 70] G. Gonenc, “A Method for the Design of Fault Detection Experiments”, IEEE Trans. on Computers, Vol. 19, No.6, June 1970.

[Goud 84] M. G. Gouda, Y. T. Yu, “Protocol Validation by Maximal Progressive Exploration”, IEEE Trans. on Comm. Vol. 32, No. 1, January 1984.

[Goum 81] H. Goumaa, D. Scott, “Prototyping as a tool in the Specification of User Requirements”, Proc. 5th IEEE Int’l Conf. Software Eng. pp. 333-342, 1981.

[Grab 93] Jens Grabowski, Dieter Hogrefe, Robert Nahm, “A Method for the Generation of Test Cases Based on SDL and MSCs”, Institut fur Informatik, Universitat Bern, April 1993.

[Grab 94] Jens Grabowski, “SDL and MSC Based Test Case Generation: An Overall View of the SaMsTaG Method”, Institut fur Informatik, Universitat Bern, May 1994.

[Guer 90] Djaffar Gueraichi, Luigi Logrippo, “Derivation of Test Cases for LAP-B form a LOTOS Specification”, Proc. Formal Description Techniques, II, pp. 361-374, 1990.

[Hare 87], David Harel, “StateCharts: A Visual Formalism for Complex Systems”, Science of Computer Programming 8, 1987.

[Higa 92] Teruo Higashino, Gregor v. Bochmann, Xiangdong Li, Keiichi Yasumoto and Kenichi Taniguchi, “Test System for a restricted Class of Lotos Expressions with Data Parameters”, Proceedings of the Fifth IFIP Workshop on Protocol Test Systems (IWPTS’92), North Holland, pp. 205-216, September 1992.

[Higu 93] M. Higuchi, H. Seki, T. Kasami, “A Method for Analyzing Communicating Finite State Machines Using Limited Reachability Analysis”, The Institute of Electronics, Information & Communication Engineers, Technical Report of IEICE, 1993.

[Hirs 85] E. Hirsh, “Evolutionary Acquisition of Command and Control Systems”, Program Manager, Nov-Dec, pp. 18-22, 1985.

[Hoar 85] C. A. R. Hoare, “Communicating Sequential Processes”, Prentice-Hall International, Englewood Cliffs, New Jersey. 1985.

[Hoff 93] D. Hoffman, P. Strooper, “A Case Study in Class Testing”, in Proc. CASCON’93, Toronto, Canada, Octobre 24-28, pp. 472-482.

[Hogr 90] D. Hogrefe, “Conformance testing based on formal methods”, in FORTE 90, Formal Description Techniques (ed. J. Quemada, A. Fernandez).

[Holz 91] Gerard J.Holzmann, “Design and Validation of Computer Protocols”, Prentice Hall, 1991.

[Holz 92] G. Holzman, P. Godefroid, D. Pirottin, “Coverage Preserving Reduction Strate-

gies for Reachability Analysis", Protocol Specification, Testing and Verification XII, IFIP, 1992.

[Howd 78] William E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing", Software-Practice and Experience, vol 8, 381-397, 1978.

[Howd 82] W. E. Howden, "Weak Mutation Testing and Completeness Tests", IEEE Transaction on software Engineering, vol. 8, No 2, pp. 371-379, Juillet 1982.

[Huan 95] Chung-Ming Huang, Yuan-Chuen Lin, Ming-Yuhe Jang, "Executable Data Flow and Control Flow Protocol Test Sequence Generation for EFSM-Specified Protocol", International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September, 1995.

[Hwan 97] I. Hwang, T. Kim, M. Jang, J. Lee, H. Oh, M. Kim "A method to derive a single-EFSM from communication multi-EFSM for data part testing", IWTCS, Chapman & Hall (1997) pp. 91-106.

[ISO 94A] International Organization for Standardization, "Conformance testing methodology and framework - part 1: general concepts", 1994.

[ISO 94B] International Organization for Standardization, "Conformance testing methodology and framework - part 2: abstract test suite specification", 1994.

[ISO 92] International Organization for Standardization, "Conformance Testing Methodology And Framework - Part 3: The Tree And Tabular Combined Notation (TTCN)", 1992.

[ISO 94C] International Organization For Standardization, "Conformance testing methodology and framework - part 4: test realization", 1994.

[ISO 94D] International Organization For Standardization, "Conformance testing methodology and framework - part 5: requirements on test laboratories and clients for the conformance assessment process", 1994.

[ISO 94E] International Organization For Standardization, “Conformance testing methodology and framework - part 6: protocol profile test specification”, 1994.

[ITEX 95] “ITEX User Manual”, Telelogic AB, 1995.

[Jack 75] M. Jackson, “Principles of Program Design” Academic Press, New York, N.Y., 1975.

[Jaff 87] J. Jaffar, J-L Lassez, “Constraint Logic Programming System”, Proceeding 14th ACM POPL Conference, Munich, January 1987.

[Jone 84] T.C. Jones “Reusability in Programming: A Survey of the State of the Art”, IEEE Trans. Software Eng., Vol. SE-10, pp. 488-494, Septembre 1984.

[King 76] James C.King, Thomas J. Watson “Symbolic Execution and Program Testing”, Communications of the ACM, vol 19, No7, July 1976.

[Koha 78] Z. Kohavi, “Switching and Finite Automata Theory”, McGRAW-HILL COMPUTER SCIENCE SERIES, 1978.

[Kore 83] B. Korel, J. Laski, “A Data Flow Oriented Program Testing Strategy”, IEEE Trans. Software Eng., SE-9 (3), pp. 347-354, 1983.

[Kore 87] B. Korel, “The Program Dependence Graph in Static Program Testing”, Information Processing Letters 24, pp. 103-108, 1987.

[Kroe 86] Fred Kroeger, “Temporal Logic of Programs”, Monographs on Theoretical Science. Springer-Verlag.

[Krop 93] P. G. Kropf, K. Guggisberg, “Traffic Simulation with SystemSpecs”, EPP 93, Oesterreichische Computergesellschaft, Vienna, Austria, February 1993.

[Ledu 91] G. Leduc, “On the Role of Implementation Relations in the Design of Distributes Systems Using LOTOS”, Thèse d’agrégation de l’enseignement Supérieur,

Université de Liege, 1991.

[Lee 96] David Lee, Krishan K. Sabnani, David M. Kristol, Sanjoy Paul, “Conformance Testing of Protocols Specified as Communicating Finite State Machines- A Guided Random Walk Based Approach”, IEEE Trans. on Comm. Vol. 44, No. 5, Mai 1996.

[Lima 97] L.P. Lima, A. R. Cavalli, “A pragmatic approach to generating test cases for Embedded Systems”, IWTCS, Chapman & Hall, pp. 288-307.

[Linn 90] Richard J. Linn Jr, “Conformance Testing for OSI Protocols”, Computer Networks and ISDN Systems, Vol. 18, pp. 203-219, 1990.

[Linn 94] Richard J. Linn, Umit Uyar, “Conformance Testing Methodologies and Architecture for OSI Protocols”, IEEE Computer Society Press, 1994.

[Loto 87] ISO DIS8807, “LOTOS: a Formal Description Technique”, 1987.

[Luo 93] G. Luo, A. Petrenko et G. v. Bochmann,” Selecting Test Sequences for Partially-specified Nondeterministic Finite State Machines”, Publication #864, Université de Montréal, 1993.

[Luo 94a] Gang Luo, Gregor v. Bochmann, Alexandre Petrenko, “Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method”, IEEE Transactions on Software Engineering, Vol. 20, No. 2, February 1994.

[Luo 94b] G. Luo, A. Das et G. v. Bochmann, “Software Testing based on SDL”, IEEE Transactions on Software Engineering, SE-20(1), pp. 72-87, Janvier 1994.

[Mack 77] A. K. Mackworth, “Consistency in Networks of Relations”, Artificial Intelligence, 8(1), pp. 99-118, 1977.

[Mann 83] Zohar Manna, Amir Pnueli, “How to Cook a Temporal Proof System for your Pet Language”, In Principles of Programming Languages, Proceedings of the 10th ACM Symposium on Principles of Programming Languages, pp. 141-154, ACM 0-89791-090-7/ 83/001/0141.

[Mann 90] Zohar Manna, Amir Pnueli, “A Hierarchy of Temporal Properties”, In Principles of Programming Languages, Proceedings of the 9th ACM Symposium on Principles of Distributed Computing, pp. 377-408, ACM Press, ACM-0-89791-404-X/90/008/3777.

[Meal 55] G.H. Mealy, “A Method for Synthesizing Sequential Circuits”, Bell System Technical Journal, Vol. 34, pp. 1045-1079, Sept. 1955.

[Mill 91] Raymond E. Miller, Sanjoy Paul, “Generating Minimal Length Test Sequences for Conformance Testing of Communication Protocols”, Proc. INFOCOM, pp. 970-979, 1991.

[Mill 92] Raymond E. Miller, Sanjoy Paul, “Generating Conformance Test Sequences for Combined Control and Data Flow of Communication Protocols”, Proceeding of Protocol Specifications, Testing and Verification (PSTV’ 92), Florida, USA, June 1992.

[Miln 89] A. J. R. G. “Communication and Concurrency”, Addison-Wesley, Reading, Massachusetts 1989.

[Moni 1985] Monien, B. “The complexity of determining paths of length k”, Proc. Int. Workshop on Graph.

[Mont 74] U. Montanari, “Networks of Constraints: Fundamental Properties and Applications to Picture Processing”, Information Science, 7(2), pp. 95-132, 1974.

[Moor 64] E. F. Moore, “Sequential Machines: Selected Papers” Addison Wesley Publishing Company, Inc., Reading, Massachusetts 1964.

[More 90] L. J. Morell, “A Theory of Fault-Based Testing”, IEEE Transactions on Software Engineering, SE-16(8).

[Mura 89] Tadao Murata, “Petri Nets: Properties, Analysis and Applications”, Proceeding of the IEEE, Vol. 77, No. 4, April 1989.

[Myer 76] G. J. Myers,” Software Reliability”, A Willey-Inter-Science Publication 1976.

[Myer 79] G. J. Myers, “The Art of Software Testing”, John Willey & Sons, 1979.

[Nait 81] S. Naito et M. Tsunoyama, “Fault Detection for Sequential Machines by Transition Tours”, Proc. FTCS, pp. 238-243, 1981.

[Nait 82] S.Naito, M.Tsunoyama, “Fault Detection for Sequential Machines by Transition Tours”, Proc IEEE, Fault Tolerant Computing Conf, 1982.

[Naur 69] P. Naur, B. randall (eds) “Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee”, NATO, Bruxelles, Belgique, 1969.

[Norr 92] M. Norris, P. Rigby “Software Engineering Explained”, John Wiley & Sons, 1992.

[Obje 96] ObjectGeode “ObjectGeode, SDL editor, User’s Guide”, Verilog 1996.

[Petr 91] A.F. Petrenko, “Checking Experiments with Protocol Machines”, Proc. of the 4th Int. Workshop on Protocol Test Systems (IWPTS), pp. 83-94, 1991.

[Petr 94] A. Petrenko, N. Yevtushenko et R. Dssouli, “Testing Strategies for Communicating FSMs”, Proc. of the International Workshop on Protocol Test Systems (IWPTS'94), Tokyo.

[Petr 96] A. Petrenko, N. Yevtushenko, G. Bochmann, R. Dssouli, “Testing in context: framework and test derivation”, a Special Issue on Protocol Engineering of Computer Communication.

[Petr 97] A. Petrenko, N. Yevtushenko, “Fault detection in Embedded Components”, IWTCS, Chapman & Hall, pp. 272-287.

[Poun 95] Dick Pountain, “Constraint Logic Programming”, Byte magazine, February 1995.

[Pres 92] R. S. Pressman, “Software Engineering a Practitioner's Approach”, McGraw-Hill, Inc., 1992.

[Pres 97] Roger S. Pressman, “Sofware Engineering: A Practitioner’s Approach”, 4th edition, McGraw-Hill Inc., 1997.

[Prob 82] R. L. Probert, “Life-cycle / Grey Box Testing”, Congressus Numerantium, volume 34, Winnipeg, Canada, 1982.

[Rafi 91] O. Rafiq, “Le test de Conformité des Protocoles”, Réseaux et Informatique Répartie, 1(1):101-142, 1991.

[Rafi 93] O. Rafiq, L. Cacciari, “La Validation Réduite des Protocoles de Communication”, Laboratoire TASC, Universite de Pau, France, Septembre 1993.

[Rama 95] T. Ramalingom, Anindya Das, K. Thulasiraman, “A Unified Test Case Generation Method for the EFSM Model Using Context Independent Unique Sequences”, International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6 September, 1995.

[Rumb 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, “Object Oriented Modeling and Design”, Prentice Hall, 1991.

[Prob 92] R. L. Probert, O. Monkewich, “TTCN: the international notation for specifying tests of communications systems”, Computer Networks and ISDN Systems, Vol. 23, pp. 417-438, 1992.

[Rama 76] C. V. Ramamoorthy, Siu-Bun F. Ho, W. T. Chan, “On the Automated Generation of Program Test Data”, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

[Rayn 87] D. Rayner “OSI Conformance Testing”, Networks and ISDN Systems, Vol. 14, pp. 79-98, 1987.

[Royc 70] W. W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques”, 1970 WESCON Technical Papers, Vol. 14, Western Electronic Show and Convention, 1970.

[Rubi 1982] J. Rubin, C. H. West, “An Improved Protocol Validation Technique”, Computer Networks, 6, April.

[Sabn 85] K.Sabnani, A.Dahbura, “A new Technique for Generating Protocol Tests”, ACM Comput. Commun. Rev. Vol 15, No 4, September 1985.

[Sabn 88] K.Sabnani, A.Dahbura, “A Protocol Test Generation Procedure”, Computer Networks and ISDN Systems, Vol.15, pp. 285-297, 1988.

[Sara 89] R.Saracco, J.R.W. Smith, R.Reed, “Telecommunication Systems Engineering using SDL”, Elsevier Science publishers B. v, 1989.

[Sari 82] Behcet Sarikaya, Gregor V. Bochmann, “Some Experience With Test Sequence Generation for Protocols”, Proc. Second Int’l Workshop Protocol Specification, Testing, and Verification, pp. 555-567, 1982.

[Sari 86] B.Sarikaya, G.v. Bochmann, “Obtaining Normal Form Specifications for Protocols”, In Computer Network Usage: Recent Experiences, Elsevier Science Publishers, 1986.

[Sari 87] B.Sarikaya, G.v. Bochmann, E. Cerny, “A Test Methodology for Protocol Testing”, IEEE Transactions on Software Engineering, Vol. 13, No. 5, pp 518-531, May 1987.

[Sari 93] B. Sarikaya, “Principles of Protocol Engineering and Conformance Testing”, Ellis Horwood Series in Computer Communications and Networking, 1993.

[Schi 97] I. Schieferdecker 1997, B. Stepien, A. Rennoch, “PerfTTCN, a TTCN language extension for performing testing”, in Testing of Communicating Systems” Vol. 10, Chapman & Hall, pp. 21-36, 1997.

[Schm 98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, B. Koch “Autolink - Putting SDL-based test generation into practice”, IWTCS’98, Tomsk, Russie, pp. 227-243, 1998.

[SDL 87] CCITT SG XI, Recommendation Z.100, 1987.

[SDT 93] “SDT Users Guide”, TELELOGIC Malmö AB, 1993.

[SDT 95] “SDT User Manual”, Telelogic Malmö AB, 1995.

[Shen 92] Y.N.Shen, F.Lombardi, A.T.Dahbura, “Protocol Conformance Testing Using Multiple UIO Sequences”, IEEE Transactions on Communications, Vol 40, No 8, August 1992.

[Shen 92A] Y.N.Shen, F.Lombardi, “Evaluation and Improvement of Fault Coverage of Conformance Testing by UIO Sequences”, IEEE Transactions on Communications, Vol 40, No 8, August 1992.

[Sidh 88] Deepinder Sidhu, Ting-kau Leung, “Fault Coverage of Protocol Test Methods”, Proceeding IEEE INFOCOM’88, March 1988.

[Sidh 89] Deepinder Sidhu, Ting-kau Leung, “Formal Methods for Protocol Testing: A Detailed Study”, IEEE Trans on Software Engineering, Vol 15, pp 413-426, 1989.

[Spiv 92] M. Spivey, “The Z Notation”, A Reference Manual, Prentice Hall, 1992.

[Toll 90] P. Tollkuehn, L. Zs. Varga, “Analysis of Formal Approaches in Protocols Conformance Testing”, KFKI- 1990-42/M, Preprint, 1990.

[Toua 96] Athmane Touag, Anne Rouger, “Génération de tests à partir de spécifications basées sur une construction partielle de l’arbre d’accessibilité”, CFIP’96, Rabat, Maroc, pp. 515-529, 1996.

[Turn 93] Kenneth J. Turner, “Using Formal Description Techniques: An Introduction to Estelle, Lotos and SDL”, John Wiley & Sons, 1993.

[Ural 86] H. Ural, An Interactive Test Sequence Generator, Proc. ACM SIGCOMM Symposium 1986, Stowe, Aug. 1986.

[Ural 87a] H. Ural, “Test Sequence Selection Based on Static Data Flow Analysis”, Com-

puter Communications, Vol.10, No.5, Octobre 1987.

[Ural 87b] H. Ural, “A Test Derivation Method for Protocol Conformance Testing”, in Proc. IFIP Symp. on Protocol Specification, Testing and Verification VII, North Holland Publ., pp. 347-358, 1988.

[Ural 91] H. Ural, B. Yang, “A Test Sequence Selection Method for Protocol Testing”, IEEE Transactions on Communication, Vol 39, No4, April 1991.

[Vuon 89] S. T. Vuong, W. W. L. Chan et M. R. Ito, “The UIOv Method for Protocol Test Sequence Generation”, In the 2-nd International Workshop Protocol Test System, Berlin, Germany, Octobre 3-6 1989.

[Vuon 90] Son T. Vuong, Kai C. Ko, “A Novel Approach to Protocol Test Sequence Generation”, Proc. of GlobalCOM’90, pp. 1880-1884, 1990.

[Vuon 91] S. T. Vuong et J. Curgus, “On Test Coverage Metrics for Communication Protocols”, In Proceedings of the IWPTS IV, Leinschendam. The Netherlands, 1991.

[Vuon 92] Son T. Vuong, Wendy W.L. Chan, et al. “One Chain of Techniques for Generating Protocol Test Sequences”, to be published.

[Wang 87] Baoyu Wang, David Hutchinson, “Protocol Testing Techniques”, Computer Communication, Vol 10, No 2, pp79-87, April 1987.

[Wass 77] A. Wasserman, “On the Meaning of Discipline in Software Design and Development”, Software Engineering Techniques, Infotech State of the Art Report, 1977.

[Wata 1981] Watanabe, “A fast algorithm for finding all shortest paths”, Inform. Process. Lett. 13, 1-3.

[Weyu 85] E.J Weyuker, S.Rapps, “Selecting Software Test Data using Data Flow Information”, IEEE Transactions on Software Engineering, April 1985.

[Weyu 88] E. J. Weyuker, “Evaluating Software Complexity Measures”, IEEE Transactions on Software Engineering, Vol. 14, No. 9, Septembre 88, pp. 1357-1365.

[Weyu 88b] E. J. Weyuker, “The Evaluation of Program-Based Software Test Data Adequacy Criteria”, Communication of the ACM, Vol. 31, No. 6, pp 668-675, 1988.

[Weyu 93] E. J. Weyuker, “More Experience with Data Flow Testing”, IEEE Transactions on Software Engineering, 19(9):912-919, Septembre 1993.

[Wiles 93] A. Wiles, A. Ek, J. Ellsberger, “Experience with Computer Aided Test Suite Generation”, in Proc. 6th Int. IFIP Workshop on Protocol Test Systems, Pau, France, 1993.

[Whit 78] C. H. White, “System Reliability and Integrity”, In Infotech State of the Art Report. Infotech, 1978.

[Wolp 89] P. Wolper, “On the relations of Programs and Computations to Models of Temporal Logic”, In B. Banieqbal, H. Barringer and A. Pnueli, editors, Proceedings Temporal Logic in Specifications, Vol. 398 of Lecture Notes in Computer Science, pp. 75-123.

[Wu 89] J. P. Wu and S. T. Chanson, “Test Sequence Derivation Based on External Behavior Expression”, Proc. of 2nd International Workshop on Protocol Test Systems, 1989.

[Yao 95] M. Yao, “On the Development of Conformance Test Suites in View of their Fault Coverage”, Thèse de doctorat, I.R.O., Université de Montréal, 1995.

[Yao 94] M. Yao, A. Petrenko et G. v. Bochmann, “Fault Coverage Analysis in Respect to an FSM Specification”, IEEE INFOCOM'94, Toronto, Canada, pp.768-775, Juin 1994.

[Yoel 90] Michael Yoeli, “Formal Verification of Hardware Design”, IEEE Computer Society Press Tutorial, pp 5-15, 1990.

[Zhou 87] Chaochen Zhou, “Specifying Communicating Systems with Temporal Logic” Institute of Software, Academia Sinica, 1987.

[Zhou 92] Limin Zhou, “A New Approach to Generating and Selecting Test Sequences for Conformance Testing”, Master Thesis, Department of Computer Science, University of British Columbia, November 1992.

[Zhu 94] J. Zhu et S. T. Chanson, “Toward Evaluating Fault Coverage of Protocol Test Sequences”, In Proceedings of the International Symposium on PSTV XIV, Vancouver, Canada, 1994.