Université de Montréal

Génération de cas de test pour les systèmes temps réel modélisés par des automates à entrées sorties temporisées

par
Abdeslam En-Nouaary

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Avril 2001

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée :

Génération de cas de test pour les systèmes temps réel modélisés par des automates à entrées sorties temporisées

présentée par :

Abdeslam En-Nouaary

a été évaluée par un jury composé des personnes suivantes :

Guy Lapalme	(président-rapporteur)
Rachida Dssouli	(directeur de recherche)
Ferhat Khendek	(co-directeur de recherche)
El Mostapha Aboulhamid	(membre du jury)
John Thistle	(examinateur externe)
Christian Léger	(représentant du doyen)

Thèse acceptée le ../../2001

Sommaire

Les systèmes temps réel interagissent avec leurs environnements en utilisant des signaux d'entrée sortie sous des contraintes temporelles. Celles-ci spécifient les plages temporelles où le système en question peut accepter une entrée ou produire une sortie. Le mauvais fonctionnement de ces systèmes est généralement dû au non respect de ces contraintes temporelles et peut engendrer des conséquences catastrophiques. Le test est l'une des techniques formelles permettant de s'assurer du bon fonctionnement des systèmes temps réel. Cette technique consiste à soumettre des cas de test à l'implantation du système et à observer ses sorties. Si ces sorties sont égales à celles attendues, l'implantation est dite conforme à sa spécification ; sinon, elle est dite non-conforme. Le modèle de spécification à utiliser pour modéliser un système temps réel dépend de plusieurs facteurs dont l'expressivité et la sémantique du temps. Si ces facteurs sont importants alors le modèle peut être complexe.

Durant les trois dernières décennies, plusieurs méthodes ont été développées pour tester les protocoles de communication. La plupart de ces méthodes se basent sur le modèle des machines à états finis (MEFs) et celui de MEF étendus (MEFEs). Les cas de test générés à partir d'une MEF permettent de détecter les fautes de sortie et les fautes de transfert. Quant au test de MEFEs, les cas de test générés permettent de tester le flux de données en se basant sur certains critères. L'objectif principal de ces méthodes est de s'assurer que les cas de test générés sont exécutables ; c'est-à-dire, les prédicats de chaque transition sont satisfaites.

Malgré les progrès importants qu'a connu le test des MEFs et des MEFEs, le test des systèmes temps réel reste un nouveau domaine de recherche. Le problème fondamental du test des systèmes temps réel est l'existence des variables d'horloges et des contraintes sur ces variables ainsi que la sémantique du temps adoptée. Lors du test d'une implantion d'un système temps réel, il ne suffit pas d'appliquer les entrées et d'observer les sorties de l'implantation mais il faut vérifier aussi si ces sorties sont produites dans les intervalles de temps permis par la spécification.

Dans cette thèse, nous passons en revue les méthodes existantes de génération de cas de test et nous présentons nos méthodes de génération de cas de test pour les systèmes temps réel modélisés par des automates à entrées sorties temporisées (AESTs). Nous présentons aussi le modèle temporel de fautes et la couverture de fautes de nos méthodes. Pour les systèmes modélisés par un seul AEST, notre méthode de génération de cas de test, appelée la méthode Wp temporisée, consiste en quatre étapes principales. Dans la première étape, nous échantillonnons le graphe de régions en utilisant une granularité qui ne dépend que du nombre d'horloges. L'objectif de cette étape est de construire un automate facilement testable appelé automate de grille. Dans la deuxième étape, nous transformons l'automate de grille en une MEF non-déterministe observable (MEFN). Dans la troisième étape, nous minimisons la MEFN en utilisant un algorithme légèrement différent de celui de la minimisation classique des MEFs. Dans la dernière étape, nous utilisons une version adaptée de la méthode Wp généralisée pour générer des cas de test temporisés à partir de la machine résultat de la troisième étape. Les cas de test générés par notre méthode détectent toutes les fautes susceptibles d'exister dans une implantation d'un AEST. Pour les systèmes modélisés par un ensemble d'AESTs communicants (AESTC), notre approche consiste à tester une composante (AESTCs) dans le contexte formé des autres composantes. D'abord, nous sélectionnons les transitions qui influencent (ou qui sont influencées par) l'exécution de la composante à tester. Ensuite, nous construisons le produit partiel du système en n'utilisant que les transitions sélectionnées pendant la première étape. Enfin, nous appliquons la méthode Wp temporisée pour

générer les cas de test à partir de l'AEST représentant le produit partiel du système. Cette méthode permet d'éviter le problème d'explosion combinatoire dû à la construction du produit complet du système.

Mots-clés : test, automates temporisés, systèmes temps réel, systèmes temps réel encapsulés, modèle de fautes, couverture de fautes.

Table des matières

Liste des abréviations

AES	Automates à entrées sorties
AEST	Automates à entrées sorties temporisées
AESTC	Automates à entrées sorties temporisées Communicants
AT	Automates temporisés
ESTELLE	Extended state transition model
GC	Graphe des contraintes
ISO	International standardization organization
IST	Implantation sous test
ITU	International telecommunication union
LOTOS	Language of temporal ordering specification
MEF	Machines à états finis
MEFC	Machines à états finis communicante
MEFD	Machines à états finis déterministe
MEFE	Machines à états finis étendus
MEFN	Machines à états finis non déterministe
MEFNO	Machines à états finis non déterministe observable
MEFTN	Machines à états finis temporisée non déterministe

MTC	Machines temporisée communicante
OSI	Open systems inteconnection
PCO	Point de contrôle et d'observation
PCT	Protocole de coordination de test
PICS	Protocol implementation conformance statements
PIXIT	Protocol implementation extra information for testing
PS	Primitive de service
PTP	Processus de temporisateurs paralleles
RP	Réseaux de Petri
SDL	Specification and description language
STE	Système de transitions étiquetées
SST	Système sous test
TDF	Technique de description formelle
TI	Testeur inférieur
TS	Testeur supérieur
TTCN	Tree and tabular combined notation
UDP	Unité de données de protocole

Table des figures

Liste des tableaux

Remerciements

J'exprime ma profonde gratitude tout d'abord à mes directeurs de recherche Rachida Dssouli et Ferhat Khendek qui m'ont proposé ce sujet et m'ont assuré un support financier tout au long de mon doctorat. Je leur suis également reconnaissant pour leurs conseils, leurs discussions et leurs disponibilités tout au long de mon travail de recherche.

Je remercie les professeurs John Thistle, Guy Lapalme, El Mostapha Aboulhamid et Christian Léger pour l'honneur qu'ils me font en faisant partie du jury de ma thèse.

Je remercie aussi tous les membres du groupe Téléinformatique qui, d'une façon ou d'une autre, ont contribué à la réalisation de cette recherche.

Je remercie également mes parents, mes frères et soeurs qui m'ont supporté et encouragé tout au cours de mes études. Je leur dédie ce travail.

Finalement, je remercie du fond du coeur ma femme Salwa qui n'a cessé de me procurer le soutien moral et affectif durant les deux dernières années de cette thèse.

Chapitre 1

Introduction

De nos jours, les systèmes logiciels sont de plus en plus utilisés dans des systèmes critiques tels que le suivi des patients, le contrôle des usines, le contrôle du trafic aérien, etc... Nous assistons aussi à une évolution des réseaux à haut débit et des applications multimédia. Tous ces systèmes, dits *systèmes temps réel*, possèdent des contraintes temporelles conditionnant leurs fonctionnements. En effet, le fonctionnement de ces systèmes n'est pas uniquement lié aux événements en entrée/sortie mais aussi aux plages de temps durant lesquelles ces événements ont lieu. De ce fait, s'assurer que ces systèmes fonctionnent correctement est une tâche difficile et complexe.

Pour faire face aux problèmes liés aux systèmes temps réel, plusieurs chercheurs se sont penchés sur la conception et le développement des méthodes efficaces afin de garantir des systèmes fiables et en réduire la complexité. Ainsi, plusieurs modèles formels ont été proposés pour l'étude des systèmes temps réel. Ils sont principalement des extensions des modèles classiques (machines à états finis, réseaux de Petri, logiques temporelles, etc...) par l'ajout de la notion de temps. Les erreurs dans les systèmes temps réel sont souvent dues au non respect des contraintes temporelles et peuvent avoir des conséquences graves[MMM95]. Il existe deux approches différentes pour améliorer la qualité d'un système logiciel, à savoir la vérification et le test.

Les techniques de vérification traitent la spécification du système à étudier et visent à s'assurer que la spécification du système satisfait certaines propriétés fonctionnelles et temporelles. Cependant, l'exactitude de la spécification d'un système ne garantit pas forcément l'exactitude de son implantation. Le test, quant à lui, est une importante activité dans le cycle de développement des systèmes informatiques. Il vise à vérifier le comportement d'une implantation d'un système par rapport à sa spécification. Autrement dit, le test a comme but de s'assurer que l'implantation d'un système respecte sa spécification. Ceci se fait par la soumission à l'implantation, communément appelée *Implantation Sous Test (IST)*, d'une séquence d'actions appelée *Suite de Test*, et la comparaison des sorties de l'IST à celles attendues. Si les sorties de l'IST sont les mêmes que celles attendues, l'IST est dite conforme à sa spécification ; sinon elle est dite non conforme.

Il existe principalement trois stratégies de test : le *test boîte blanche*, le *test boîte noire* et le *test boîte grise*. Dans le test boîte blanche, la structure interne de l'implantation est connue et la suite de test est générée à partir de cette structure. Dans le test boîte noire, la structure interne de l'implantation n'est pas connue ; on utilise la spécification du système pour générer la suite de test. Finalement, le test boîte grise est un peu différent du test boite noire dans le sens où on suppose que la structure modulaire de l'implantation est connue mais non les détails des programmes de chaque composante.

Contrairement au test des systèmes non temporisés, le test des systèmes temps réel consiste à vérifier non seulement les réactions de l'implantation aux stimulis qui lui sont soumis mais aussi le temps de ces réactions. Ce qui implique que les méthodes existantes de génération de cas de test ne sont pas applicables aux systèmes temps réel. Afin de répondre aux besoins du test de ces systèmes, il faut, si possible, adapter les méthodes existantes sinon en développer de nouvelles.

1.1 Objectifs de cette thèse

Dans le cadre de cette thèse, nous proposerons de développer des méthodes de génération de cas de test pour les systèmes temps réel afin de décider si une implantation est conforme à sa spécification ou non. Les systèmes que nous étudierons sont modélisés par des automates à entrées sorties temporisées (AEST) pour décrire la communication entre le système et son environnement ainsi que les contraintes temporelles régissant cette communication. Les AESTs sont une variante des automates temporisés d'Alur et Dill [AD94] dans lesquels l'alphabet est subdivisé en un ensemble d'entrées représentant les messages que le système reçoit de son environnement et un ensemble de sorties représentant les messages que le système produit à son environnement.

Comme nous l'avons souligné précédemment, le test des systèmes temps réel diffère de celui des systèmes classiques (non temporisés) à cause des contraintes temporelles dans leur comportement. De ce fait, les méthodes de génération de cas de test à partir d'une machine à états finis (étendus) ne sont pas directement applicables à un automate à entrées sorties temporisées. Il faut alors étendre le fondement du test non temporisé pour prendre en considération la notion et la sémantique de temps dans les systèmes temps réel. Dans le cas où le système est modélisé par plusieurs AESTs communicants les uns avec les autres et avec l'environnement, les problèmes sont plus nombreux. D'une part, les méthodes existantes de génération de cas de test ne sont plus applicables, et d'autre part, on doit non seulement prendre en considération la relation entre les différentes composantes (AESTs) du système mais aussi éviter l'explosion combinatoire (ou explosion d'états) due à la composition des différentes composantes en un AEST global.

Notre but est de développer des méthodologies nouvelles et pratiques pour tester les systèmes temps réel modélisés par un ou plusieurs automates à entrées sorties temporisées. Plus précisément, nous visons à :

- Présenter la problématique du test des systèmes temps réel.
- Étudier le modèle de fautes des systèmes temps réel modélisés par un AEST.
- Développer une méthodologie nouvelle et pratique pour générer des cas de test à partir d'un AEST.
- Évaluer la couverture de fautes de notre méthodologie par rapport au modèle de fautes présenté.
- Tester une composante d'un système temps réel modélisé par plusieurs automates à entrées sorties temporisées communicants (AESTC) entre eux et avec l'environnement.
- Étudier le modèle de fautes dans le contexte d'un AESTC et s'intéresser à la propagation des fautes entre les différentes composantes.

1.2 Organisation de la thèse

Le reste de cette thèse est organisé de la manière suivante. Au chapitre 2, nous présentons en détail la méthodologie OSI pour le test de conformité et nous introduisons, d'une manière générale, les différents éléments utilisés dans la pratique du test de conformité tout en montrant leurs utilités. Ces éléments sont la spécification formelle, la relation de conformité, les hypothèses de test, le modèle de fautes et la couverture de fautes.

Au chapitre 3, nous parlons de la spécification formelle des systèmes dans un cadre général incluant les systèmes temps réel et nous présentons les principaux modèles utilisés à cette fin. Ainsi, nous introduisons les machines à états finis, les machines à états finis étendus, les techniques de description formelle (SDL, LOTOS, ESTELLE), les automates temporisés, etc... Pour chacun de ces modèles, nous donnons les caractéristiques et les propriétés qui nous seront utiles dans les chapitres suivants.

Au chapitre 4, nous présentons l'état de l'art de la génération de cas de test à

partir des modèles formels. Nous passons en revue la littérature pertinente à notre travail à savoir les méthodes de génération de cas de test à partir des MEFS, du test des MEFEs et du test temporisé.

Au chapitre 5, nous présentons en détail le modèle des automates à entrées sorties temporisées (AESTs) et nous discutons les problèmes liés au test des systèmes temps réel modélisés par des AESTs tout en le comparant au test des MEFs et des MEFEs.

Au chapitre 6, nous introduisons notre modèle de fautes temporelles basé sur les AESTs. Nous illustrons chaque faute par un exemple et nous donnons son effet sur l'exécution du système.

Au chapitre 7, nous présentons notre approche de génération de cas de test pour les systèmes temps réel modélisés par un seul AEST. Nous étudions aussi la couverture de fautes de notre méthode en prouvant qu'elle est complète.

Au chapitre 8, nous étudions le test des systèmes encapsulés modélisés par plusieurs AESTs communicants entre eux et avec l'environnement. Nous révisons d'abord le modèle de fautes présenté dans le chapitre 6, dans le contexte d'un AESTC en examinant l'effet du contexte sur le test et la propagation de fautes entre les différentes composantes du système. Ensuite, nous présentons notre méthodologie pour tester un AESTC du système dans le contexte formé des autres AESTCs.

Finalement, le chapitre 9 résume cette thèse et présente des extensions possibles.

Chapitre 2

Test de conformité

Ce chapitre présente les différents concepts du test de conformité. Nous commençons par introduire la méthodologie OSI. Ensuite, nous présentons les éléments nécessaires pour l'automatisation de cette méthodologie. Nous verrons plus particulièrement les notions de spécification formelle, de relations de conformité, du modèle de fautes, des hypothèses de test, et de couverture de fautes.

2.1 La méthodologie OSI

Le test de conformité se base sur la norme ISO/IEC IS9646[ISO91a, Ray87, Tre92a, Nah95]. Cette norme définit une méthodologie du test de conformité des protocoles de communication spécifiés dans un langage naturel. Bien qu'elle ait été développée initialement pour les protocoles OSI, la norme ISO9646 peut être utilisée pour tester tout autre protocole ou système. Le test de conformité vise à prouver qu'une implantation d'un protocole est conforme à sa spécification de référence. Ceci consiste à vérifier que l'implantation du protocole ne présente que le comportement permis par la spécification. Cependant, pour des raisons pratiques et économiques, il est impossible de tester exhaustivement le comportement d'une implantation d'un protocole. C'est pourquoi le test de conformité peut seulement montrer la présence des erreurs et pas leur absence. La Figure 2.1 montre la méthodologie OSI, conjointe-

ment avec le processus d'implantation d'un protocole [Tre92a, Ray87, Nah95]. Dans ce qui suit, nous présentons les différents éléments constituant cette méthodologie.

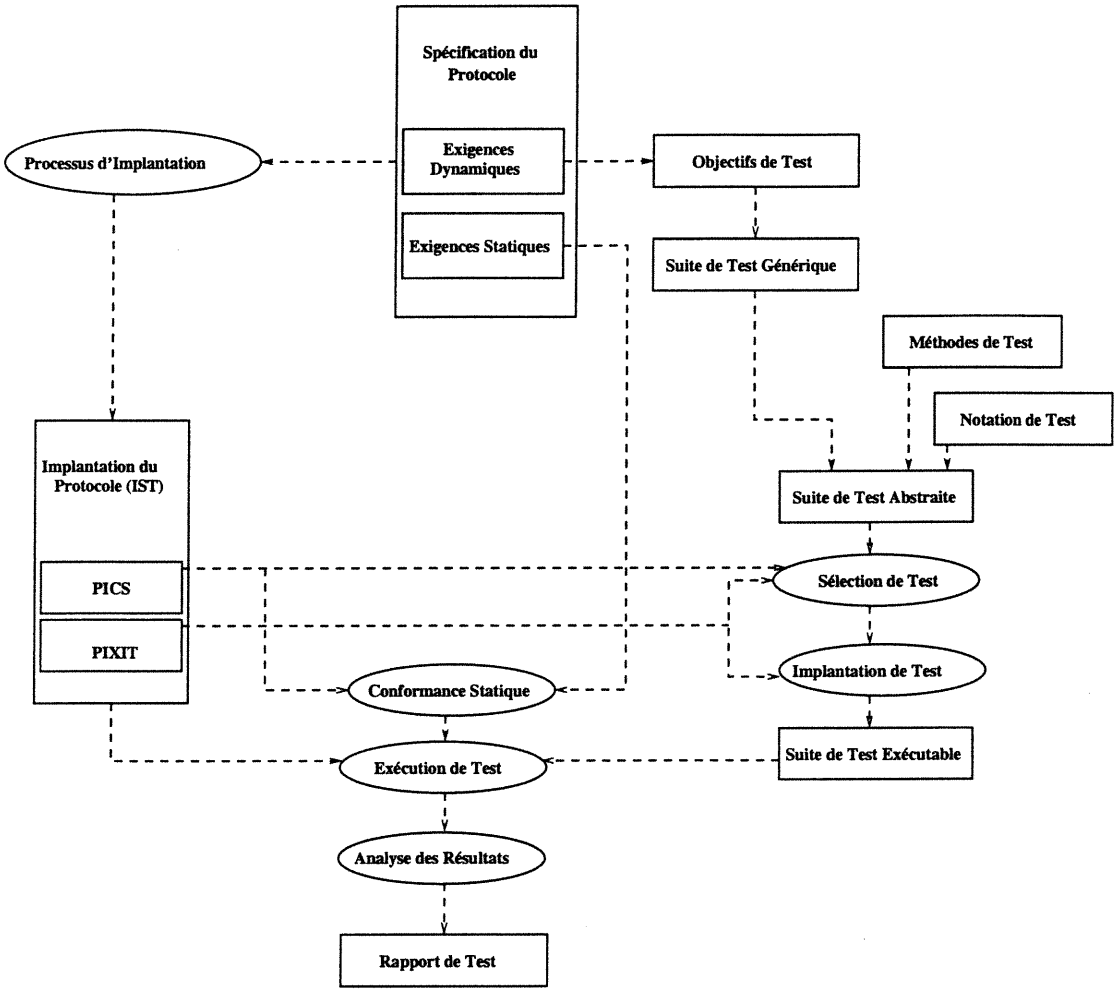

FIG. 2.1: L'activité du test de conformité

2.1.1 Spécification et exigences de conformité

Une spécification d'un protocole de communication est un ensemble d'exigences de conformité qui doivent être respectées par toute implantation de ce protocole. Les exigences de conformité sont de deux types, les *exigences de conformité statiques* et les *exigences de conformité dynamiques.* Les exigences statiques définissent les fonctions minimales qu'une implantation doit fournir afin de faciliter la communication.

Ces exigences peuvent être spécifiées soit à un haut niveau tel que le groupement des fonctionnalités dans des classes de protocoles, soit à un niveau détaillé tel que la spécification du domaine d'un paramètre ou d'un temporisateur. Les exigences dynamiques, quant à elles, spécifient le comportement observable du protocole en ce qui concerne la communication avec l'environnement. Elles définissent le comportement observable d'une entité de protocole à ses interfaces en termes d'unités de données de protocole (UDP) et de primitives de service (PS) échangées, le format des UDPs, la table des transitions, et la relation entre le contenu des différentes UDPs.

Afin de tester une implantation, son fournisseur doit communiquer au laboratoire de test un certain nombre d'informations concernant l'implantation. Ces informations sont données sous forme de deux documents appelés *PICS (Protocol Implementation Conformance Statements)* et *PIXIT (Protocol Implementation eXtra Information for Testing)*. Le PICS définit la liste des fonctionnalités et des options qui ont été implantées et la liste des propriétés qui ont été omises. Le PIXIT, quant à lui, contient des informations supplémentaires nécessaires au test de l'implantation. Parmi ces informations, il y a celles qui sont liées à la méthode de test utilisée pour exécuter les cas de test, celles concernant la réalisation du testeur supérieur, et celles spécifiant le domaine des valeurs de certains paramètres ou temporisateurs. Une implantation est dite conforme à sa spécification si elle satisfait en même temps les exigences de conformité statiques et dynamiques tout en prenant en compte les fonctions et les options de PICS.

2.1.2 Génération de tests

La première étape dans la procédure du test de conformité est la génération ou la dérivation de tests. Elle consiste à dériver systématiquement les cas de test à partir de la spécification de référence. L'objectif est de développer une méthode efficace pour générer une suite de test abstraite ; c'est-à-dire, indépendante de l'implantation. Cette suite de test doit être capable de tester tous les aspects du protocole à un niveau

suffisant de détail. Elle est généralement écrite dans le langage de représentation de tests TTCN (Tree and Tabular Combined Notation). Une suite de test abstraite est composée de plusieurs cas de test abstraits. Ces cas de test sont hiérarchiquement structurés en des groupes de test selon leurs aspects logiques. Dans cette thèse, nous n'allons pas donner les cas de test en TTCN.

La génération des cas de test se basent sur un ensemble d'*objectifs de test* extraits à partir de la spécification. Normalement, seules les exigences de conformité dynamiques sont considérées pour le développement des objectifs de test. La conformité vis à vis des exigences de conformité statiques est vérifiée par l'examination du PICS.

Un objectif de test décrit d'une manière précise la fonctionnalité qu'on veut tester. En général, à chaque fonctionnalité du protocole, on fait correspondre un objectif de test. La suite de test abstraite est développée à partir de la spécification de protocole et les objectifs de test. Pour chaque objectif de test, un *cas de test générique* est dérivé. Un cas de test générique est une réalisation de l'objectif de test sans considérer la méthode de test à utiliser ou l'environnement dans lequel sera exécuté le test. Il est composé de trois parties : le préambule, le corps de test, et le postambule. Le préambule est une suite de transitions qui commence à un état stable, généralement l'état initial, et se termine à un état où le corps de test peut être appliqué. Le corps de test est la réalisation de l'objectif de test. Le postambule est une suite de transitions qui permet de ramener le système à un état stable, généralement l'état initial, après l'application du corps de test. Les cas de test génériques sont transformés en cas de test abstraits en choisissant une méthode de test et en tenant compte de toute restriction imposée par l'environnement de test.

2.1.3 Méthodes de test

Les méthodes de test définissent la manière dont sera testée l'implantation. Elles spécifient le modèle nécessaire à l'exécution des cas de test. Ce dernier comprend

l'implantation sous test, le système sous test, le fournisseur de service sous-jacent, le testeur, et la procédure de coordination de test. L'implantation sous test (IST) est l'implantation du protocole qu'on veut tester. Elle n'est accessible que par ses interfaces inférieures et supérieures, appelées points de contrôle et d'observation (PCO). Le système sous test (SST) est le système dont fait partie l'IST. Le testeur est le système qui contrôle et observe les points d'accès de l'IST. La procédure de coordination de test (PCT) est l'ensemble des règles qui régissent le test de l'IST. Le fournisseur de service sous-jacent fait partie en même temps du ST et du SST et leur offre un moyen (une connexion) pour communiquer.

On distingue généralement deux types d'architectures de test pour le test de conformité des protocoles de communication. Ces architectures sont le *test local* et le *test externe.* Le test local correspond au test classique des logiciels, dans lequel le testeur et l'IST font partie d'un même système (même machine). Quant au test externe, le testeur et l'IST appartiennent à deux systèmes différents. Il existe trois méthodes du test externe : le *test distribué*, le *test coordonné* et le *test à distance.* La Figure 2.2 montre le test distribué où le testeur est subdivisé en deux parties, le *testeur supérieur (TS)* et le *testeur inférieur (TI)*, qui accèdent respectivement aux interfaces supérieure et inférieure (PCOs) de l'IST. L'interface inférieure est indirectement accessible à distance via le service de communication sous-jacent. Le test coordonné est similaire au test distribué sauf que le TS et le TI coopèrent entre eux, selon certaines règles (*protocole de coordination de test PCT*), à travers un canal de communication. Le test à distance correspond au test distribué où seulement le TI est utilisé (le TS est optionnel)[LDB+93].

2.1.4 Implantation de test

Cette phase consiste à transformer les cas de test abstraits en cas de test exécutables en prenant en considération les spécificités de l'IST. Ceci est fait en deux étapes : la sélection et la paramétrisation. La sélection consiste à choisir parmi les cas de tests

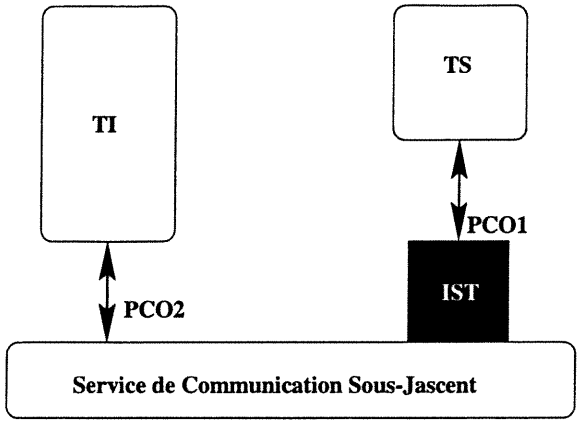

FIG. 2.2: L'architecture du test distribué

abstraits seuls ceux qui sont cohérents avec les options et les fonctionnalités mentionnées dans le PICS. La paramétrisation, quant à elle, consiste à adapter les cas de test, sélectionnés dans l'étape précédente, à l'IST et son environnement en prenant en considération les informations citées dans le PIXIT. Il s'agit, entre autre, d'instancier les variables et les temporisateurs par les valeurs fournies. Le résultat final de cette phase est un ensemble de cas de test susceptibles d'être soumis à l'IST pour conclure un verdict sur sa conformité par rapport à sa spécification.

2.1.5 Exécution de test

Cette phase consiste à appliquer les cas de test exécutables à l'IST et à observer son comportement. Les événements des cas de test exécutables sont les signaux soumis à ou reçus de l'IST. Ils sont répartis entre les testeurs inférieur et supérieur selon le PCO où ils sont appliqués ou observés. Les réactions (sorties) de l'IST sont observées par les testeurs et comparées aux réactions attendues (obtenues à partir de la spécification). En se basant sur cette comparaison, un verdict est affecté à chaque cas de test. Il peut être soit *pass*, soit *fail* ou soit *inconclusif*. *Pass* signifie que le test a réussi et que l'objectif de test qui est à la base du cas de test a été atteint. *Fail* indique que l'IST n'est pas conforme à sa spécification. Autrement dit, l'objectif de test correspondant au cas de test dont le verdict est fail n'a pas été atteint. Le verdict *inconclusif*, quant

à lui, signifie que la non conformité de l'IST ne peut être prouvée mais l'objectif de test n'a pas été atteint.
Finalement, en combinant les verdicts de tous les cas de test, l'IST est dite conforme à sa spécification si et seulement si aucun verdict *fail* n'a été assigné. Tous les détails sur le test effectué sont documentés et rapportés dans un rapport de test, appelé PCTR (Protocol Conformance Test Report).

2.2 Automatisation du test de conformité

La méthodologie OSI, résumée dans la section précédente, présente le test de conformité d'une manière informelle. Pour automatiser le processus du test de conformité, la formalisation de la méthodologie OSI est primordiale. Les éléments essentiels pour cette formalisation sont la modélisation formelle des spécifications et des implantations, la relation de conformité, le modèle de fautes, les hypothèses de test, et la couverture de test. Dans cette thèse, nous ne nous intéressons qu'à la conformité dynamique et plus particulièrement à la génération des cas de test.

2.2.1 Modèles formels de spécification

Afin de générer automatiquement des cas de test, on utilise des représentations formelles pour la spécification et l'implantation. Ces représentations sont des formalismes mathématiques qui donnent une vision claire des fonctionnalités d'un système. Elles définissent qu'est ce que le système doit faire et qu'est ce qu'il ne doit pas faire. Le comportement d'un système est constitué de plusieurs aspects dont les plus importants sont :

- l'aspect contrôle qui spécifie l'ordre causal des événements échangés entre le système et son environnement. Par exemple, à tout événement en entrée, le système doit répondre par un certain événement en sortie.
- l'aspect flux de données qui spécifie le chemin de données dans le système. Les données peuvent être soit définies dans le système soit reçues de l'environnement

comme paramètres d'un message.

- l'aspect temporel qui spécifie les contraintes temporelles entre les événements reçus et/ou produits par le système.

Ces aspects peuvent être spécifiés en utilisant les formalismes et les langages suivants :

- Les machines à états finis (MEF),
- Les réseaux de Petri (RP),
- Les machines à états finis étendus (MEFE),
- Le système de transitions étiquetées (STE),
- Les automates à entrées sorties (AES),
- Les automates temporisés (AT),
- Les techniques de description formelles (TDF) comme SDL, LOTOS, ESTELLE,
- etc...

Le choix du modèle à utiliser pour la spécification d'un système est une question qui se pose souvent dans toute étude formelle. En général, on choisit le modèle qui permet de décrire tous les aspects pertinents du système en question. Par exemple, les MEFs sont convenables pour modéliser l'aspect contrôle. Les MEFEs sont utilisés pour spécifier en même temps l'aspect contrôle et l'aspect données. Les ATs sont les plus convenables pour modéliser l'aspect contrôle et l'aspect temporel des systèmes.

2.2.2 Relations de conformité

L'objectif du test de conformité est de vérifier une certaine relation de conformité entre l'implantation et la spécification d'un système. Cette conformité est généralement exprimée par une relation mathématique, mettant en jeu le comportement de l'IST et celui de la spécification. Pour formaliser le problème, nous supposons que la spécification S et l'implantation I sont décrites dans un même modèle formel $MODF$. Les comportements de la spécification S et de l'implantation I, notés respectivement C_S et C_I, sont alors des sous-ensembles du langage décrit par $MODF$. Pour la

spécification, un comportement est un ensemble de propriétés valides. Par contre, le comportement de l'implantation est l'ensemble de propriétés satisfaites par celle-ci. La relation de conformité, que l'on note par $RelConf$, entre I et S est définie sur l'ensemble $MODF \times MODF$ de la manière suivante : $I\ RelConf\ S$ si et seulement si $\forall c \in C_S, c \in C_I$.

À partir de cette définition, il est clair qu'une relation de conformité dépend fortement du modèle formel choisi et de la définition de la notion du comportement. Autrement dit, différentes relations de conformité peuvent être obtenues en faisant varier $MODF$ et les propriétés à tester. Dans la littérature, plusieurs relations ont été utilisées. Pour les MEFs, l'équivalence de traces est souvent utilisée. Pour les STEs, la relation de bisimulation est parfois choisie. Pour les autres modèles, une adaptation de ces deux relations est souvent adoptée.

La relation de conformité est un élément important mais reste insuffisant pour conclure à une relation entre une implantation et sa spécification. Généralement, la relation de conformité est utilisée conjointement avec certaines conditions et hypothèses sur l'implantation et la spécification pour limiter l'univers de test. La Figure 2.3 clarifie davantage l'utilisation de la relation de conformité et d'hypothèses de test.

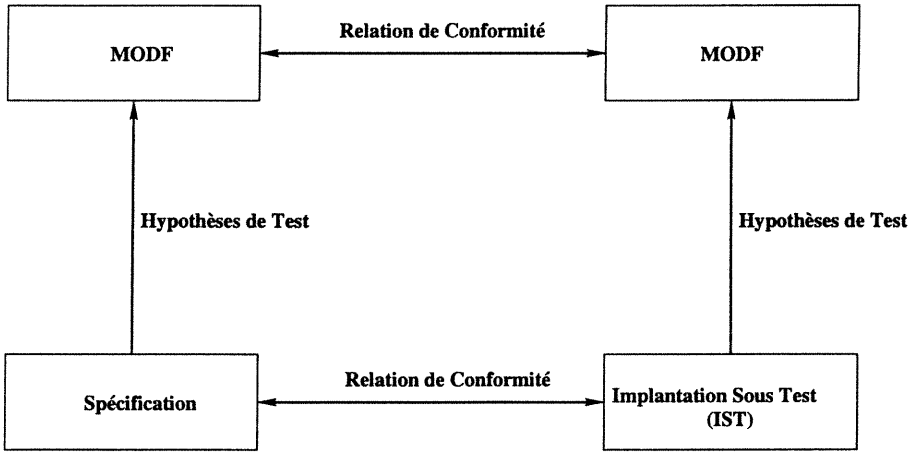

FIG. 2.3: La relation de conformité

2.2.3 Hypothèses de Test

Les hypothèses de test sont introduites pour simplifier l'activité de test. Certaines hypothèses sont implicites ; d'autres doivent être explicitement formulées. L'objectif principal de l'utilisation des hypothèses de test est la réduction de l'univers des implantations à considérer dans le test. Les hypothèses de test suivantes sont souvent utilisées dans tous les modèles lors de la génération des cas de test :

- la régularité : Cette hypothèse permet de limiter le test à un ensemble fini de comportements pour les systèmes exhibant un nombre infini de comportements. Cette hypothèse est souvent utilisée lorsqu'on traite un système dont la spécification contient des boucles simples ou même multiples. On peut tester de telles spécifications en ne traversant une boucle qu'un nombre fini de fois.
- l'uniformité : Cette hypothèse permet de regrouper les comportements similaires d'un système en des classes d'équivalence et de choisir un représentant pour chaque partition. C'est notamment le cas où une spécification contient des messages paramétrés par des variables entières ou réelles, ou des contraintes sur des variables temporelles dont le domaine est dense ou discret non limité. On peut tester ces spécifications en ne prenant qu'un nombre limité de valeurs pour ces variables.
- l'indépendance : Cette hypothèse permet de faciliter le test d'un système complexe, formé de plusieurs modules, en testant chaque composante séparément. C'est une hypothèse controversée qui est parfois omise car elle suppose qu'une erreur dans un module ne peut aucunement affecter la détection des fautes dans les autres modules. Ceci n'est pas toujours vrai du fait que les modules d'un système complexe sont généralement interdépendants et peuvent communiquer entre eux en partageant des variables ou en échangeant des messages.
- l'équité : Cette hypothèse est utilisée pour simplifier le test des systèmes non-déterministes. Plusieurs systèmes sont de nature non-déterministe. Par exemple, le parallélisme dans les systèmes distribués introduit *l'entrelacement* des actions dans les différentes composantes du système. L'hypothèse d'équité permet de

conclure que tous les choix possibles d'une implantation sont faits après un nombre raisonnable d'essais.

En plus de ces hypothèses générales, d'autres hypothèses sont également formulées sur la spécification et la structure interne de l'IST. Ces hypothèses sont nécessaires à l'application des méthodes de génération de cas de test et visent à réduire le comportement possible de l'IST afin de développer des critères de couverture de test. Par exemple, on peut supposer que l'IST est modélisée par un modèle formel avec certaines restrictions (le nombre d'états est borné supérieurement par un entier m, le modèle est complètement spécifié, le déterminisme, etc...). Nous verrons ce type particulier d'hypothèses dans les chapitres 4, 7 et 8.

2.2.4 Modèle de fautes

L'objectif principal de test de conformité est la détection des fautes dans une implantation d'un système. Une erreur est la conséquence directe ou indirecte de l'exécution d'une faute (défectuosité dans une composante du système). Pour tester le système, on doit donc tenir compte de différentes fautes possibles. Ces dernières peuvent être nombreuses et complexes. De plus, plusieurs défectuosités peuvent engendrer la même erreur pour un ou plusieurs cas de test. Pour traiter cet aspect, on regroupe souvent les fautes dans des modèles de fautes selon leurs natures et les erreurs qu'elles engendrent [BDD+91]. Un modèle de fautes permet de décrire l'effet des défectuosités à un niveau élevé d'abstraction (transfert de registres, blocs fonctionnels, etc...). Ceci implique divers compromis entre la précision et la facilité dans la modélisation et l'analyse de test. Si le modèle de fautes décrit les fautes d'une manière précise, alors un nombre limité de cas de test sera suffisant pour détecter toutes les fautes du modèle de fautes. Ceci a l'avantage de réduire le nombre de possibilités à considérer lors de la génération des cas de test.

Le modèle de fautes est très dépendant du modèle de spécification utilisé. En effet, pour chaque modèle formel de spécification il existe un modèle de fautes plus ou

moins riche pour décrire toutes les fautes potentielles d'une implantation du système. Malgré cette différence dans l'expressivité de différents modèles de fautes, les fautes 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, fautes de violation de certaines contraintes, etc...
- les fautes de données : fautes de type ou de représentation du format de données, fautes d'initialisation ou du domaine de données, fautes de référence à des variables indéfinies, etc...

Malgré que le nombre de toutes ces fautes reste inconnu et que les modèles de fautes ne couvrent qu'une partie de ces fautes, ces derniers sont très utilisés dans l'activité de test. Ils permettent, entre autres, de développer des méthodes pratiques (c'est-à-dire, non coûteuses) de génération de tests et de comparer ces méthodes selon leurs capacités à couvrir les fautes.

2.2.5 Couverture de Fautes

La couverture de fautes est un aspect important dans le processus du test de conformité. Elle permet de juger l'efficacité et la qualité d'un test. Cette dernière est mesurée par le nombre et le type des fautes détectables par le test. La couverture de fautes est une relation qui relie le modèle de fautes choisi et les cas de test générés (ou simplement la méthode de génération de test utilisée). Pour chaque faute appartenant au modèle de fautes, on regarde s'il existe un cas de test qui la détecte. La couverture de fautes d'une méthode de génération de test (ou d'une suite de test) peut être alors définie quantitativement comme étant le nombre de fautes détectées sur le nombre total de fautes recensées dans le modèle de fautes. Ainsi, on dit qu'une méthode de génération de test a une couverture complète de fautes si la suite de test générée par cette méthode détecte toutes les fautes recensées; sinon, elle est dite avoir une couverture partielle de fautes. À partir de là, il est clair que la couverture de fautes peut permettre la comparaison entre les méthodes de génération de test. Ainsi, une

méthode A est dite plus efficiente qu'une méthode B si sa couverture de fautes est meilleure que celle de B ; c'est-à-dire, A détecte plus de fautes que B. Des techniques de calcul et d'évaluation de la couverture de fautes ont été présentées dans [Yao95].

En plus de la couverture de fautes que nous venons de définir, il existe d'autres notions de couvertures de test [Cha97]. Ces dernières évaluent la qualité d'un test en se basant soit sur la spécification du système soit sur les hypothèses de test utilisées. Pour la couverture basée sur la spécification, on définit quantitativement la qualité de test par le pourcentage des éléments couverts par le test. Ces éléments sont, entre autres, les instructions, les branches et les chemins. Quant à la couverture basée sur les hypothèses de test, on la définit qualitativement par l'ensemble des hypothèses à faire sur l'implantation pour que le test soit complet.

2.3 Conclusion

Dans ce chapitre, nous avons introduit le test de conformité des systèmes logiciel. Nous avons vu que l'objectif du test de conformité est de s'assurer que l'implantation d'un système fait correctement ce pourquoi le système est conçu. Nous avons rappelé la norme ISO/IEC IS 9646. Cette norme, même si elle est informelle, est à la base du test de conformité de n'importe quel système. Pour automatiser le processus du test de conformité, nous sommes passés à la formalisation de la méthodologie OSI en donnant les différents éléments utilisés dans la pratique. Nous avons montré que l'utilisation des modèles formels pour représenter la spécification et l'implantation est le point de départ de toute formalisation du test de conformité. Ces modèles permettent de donner une vision claire de différents aspects du comportement d'un système. Nous avons aussi parlé de la relation de conformité entre une implantation et sa spécification, et nous l'avons défini, d'une manière générale, comme une relation mathématique sur le modèle formel choisi. Comme l'espace des implantations est infini, nous avons vu l'utilité des hypothèses de test et nous en avons cité quelques unes. Nous avons aussi expliqué l'importance du modèle de fautes en montrant qu'il

permet de développer des méthodes pratiques de génération de tests surtout lorsque l'application d'une relation de conformité s'avère difficile ou impossible. Enfin, nous avons vu que la couverture de fautes est un moyen de juger la qualité d'un test, permettant ainsi de comparer différentes techniques de génération de tests.

Dans le chapitre suivant, nous nous concentrerons sur les modèles et les langages formels de spécification. Nous définirons les différents modèles existant dans la littérature et nous étudierons les caractéristiques de chacun d'entre eux.

Chapitre 3

Modèles formels de spécification

Nous avons vu dans le chapitre précédent que l'utilisation des modèles formels est le point de départ de l'automatisation et la formalisation du test de conformité. Nous avons aussi cité les différents aspects constituant le comportement d'un système informatique. Dans le présent chapitre, nous allons passer en revue les différents modèles formels utilisés pour décrire le comportement des systèmes informatiques. Ces modèles diffèrent selon leurs expressivités et leurs complexités. Selon notre domaine d'intérêt, nous classons ces modèles suivant leurs expressivités en deux catégories : les modèles non temporels de spécification et les modèles temporels de spécification.

3.1 Modèles non temporels de spécification

Nous entendons par modèles non temporels de spécification les modèles et techniques qui ne permettent pas de décrire suffisamment l'aspect temporel du comportement d'un système. Dans cette catégorie de modèles et techniques formels, nous trouvons les machines à états finis, les systèmes de transitions étiquetées, les machines à états étendus, les réseaux de Pétri et les langages de description formelle (SDL, ESTELLE et LOTOS). Ces modèles ont servi pendant les trois dernières décennies à l'étude des systèmes informatiques et en particulier les protocoles de communication.

3.1.1 Les machines à états finis

Durant les trois dernières décennies, les machines à états finis (MEF) étaient le modèle le plus utilisé pour la génération des suites de test. Ceci est dû principalement à l'existence d'une théorie très élaborée pour ce modèle [Koh78, Gil61]. Une MEF est un modèle abstrait de description du comportement d'un système comme étant une séquence d'événements survenant à des moments discrets $t = 1, 2, 3, etc...$ Ces événements sont classés en deux catégories : *les entrées* et *les sorties.* Les entrées sont la stimulation de l'environnement au système décrit par la MEF alors que les sorties sont les réponses immédiates du système à ces entrées.

Formellement, une MEF est un septuplet $M = (I, O, S, s_0, D, \delta, \lambda)$, où :

- I est un ensemble fini d'entrées,
- O est un ensemble fini de sorties,
- S est un ensemble fini d'états,
- $s_0 \in S$ est l'état initial,
- $D \subseteq S \times I$ est le domaine de spécification,
- $\delta : D \longrightarrow S$ est la fonction du transfert,
- $\lambda : D \longrightarrow O$ est la fonction des sorties.

La fonction du transfert et la fonction des sorties caractérisent le comportement de la MEF. Une transition de l'état s_i à l'état s_j sur une entrée/sortie i/o est représentée par $(s_i, i/o, s_j)$ et notée par $s_i \xrightarrow{i/o} s_j$, où : $\delta(s_i, i) = s_j$ et $\lambda(s_i, i) = o$.

Une séquence d'entrées $i_1...i_k \in I^*$ est dite *acceptable* à l'état s_i, s'il existe k états $s_1...s_k \in S^k$ et une séquence de sorties $o_1...o_k \in O^*$ tel que $s_i \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \xrightarrow{i_3/o_3} ... \xrightarrow{i_{k-1}/o_{k-1}} s_{k-1} \xrightarrow{i_k/o_k} s_k$. I^*_{si} est l'ensemble de toutes les séquences acceptables en s_i et $I^*_M = I^*_{s0}$ est l'ensemble de toutes les séquences acceptables par l'automate en son état initial s_0. Une *trace* d'un état s_i est une séquence d'entrées/sorties formée d'une séquence d'entrées acceptable à s_i et de sa séquence de sorties. Nous désignons par

Traces(s_i) l'ensemble des traces de l'état s_i, et par *Traces(M)* l'ensemble des traces de la machine M (Traces(M)=Traces(s_0)).

Les fonctions du transfert et de sorties peuvent être étendues sur $D_M = \cup_{s_i \in S}(s_i \times I^*_{si})$ tout en gardant la même appellation, c'est-à-dire, $\delta : D_M \longrightarrow S$ et $\lambda : D_M \longrightarrow O^*$. Une MEF M est dite *déterministe* (MEFD) si $|\delta(s_i, \alpha)| = 1$ et $|\lambda(s_i, \alpha)| = 1$ pour tout $(s_i, \alpha) \in D_M$; sinon la machine M est dite *non-déterministe* (MEFN). Dans ce dernier cas, on distingue entre le *non déterminisme observable* et le *non déterminisme non observable*. Une MEFN M est dite *observable* (MEFNO) si et seulement si pour tout $s_i \in S$ et tout $i/o \in I \times O$ tel que $s_i \xrightarrow{i/o} s_j$ et $s_i \xrightarrow{i/o} s_k$, on a $s_j = s_k$. Nous signalons que toute MEFN *non observable* peut être facilement transformée en une MEFNO.

Il est à noter que si le domaine de spécification $D = S \times I$, alors la fonction du transfert δ et la fonction des sorties λ sont définies pour toutes les combinaisons du produit cartésien des états et des entrées (c-à-d $S \times I$). Dans ce cas, la machine est dite *complètement spécifiée*. Dans le cas contraire, la machine est dite *partiellement spécifiée*. De même, une MEF est dite *fortement connexe* si pour toute paire d'états s_i et s_j, il existe un chemin de transitions qui mène de s_i à s_j.

La Figure 3.1 montre une MEF déterministe fortement connexe et complètement spécifiée.

Pour une MEF M, deux états s_i et s_j sont dits *équivalents* si $Traces(s_i) = Traces(s_j)$; sinon ils sont dits *distinguables*. Si tous les états de M sont distinguables deux à deux alors M est dite *réduite* ou *minimale*. Deux machines M et M' sont dites équivalentes si leurs états initiaux sont équivalents. Cette relation *d'équivalence* entre les MEFs est appelée *équivalence de traces*.

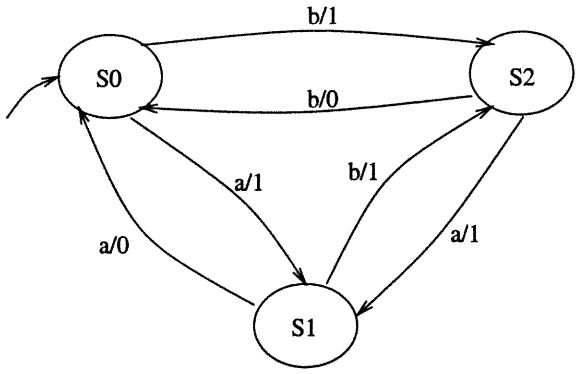

FIG. 3.1: Un exemple de MEF

3.1.2 Les Machines à états finis communicantes

L'aspect contrôle des programmes concurrents, surtout dans le domaine des protocoles de communication, peut être modélisé par un système de machines à états finis communicantes (MEFC) [LBP94, BDA96] où les MEFs communiquent entre elles à travers des files FIFO (le premier arrivé est le premier servi) et des canaux. Cette communication se fait généralement d'une manière asynchrone comme dans le cas du langage SDL.

Un système de MEFCs, noté $(M_1, M_2, ..., M_n)$, est constitué d'un certain nombre de MEFCs, $M_1, M_2, ..., M_n$, avec les propriétées suivantes :

- Chaque MEFC est munie d'une file FIFO où les entrées de la machine sont stockées et d'où elles sont retirées.
- À chaque paire de MEFCs sont associés deux canaux FIFO pour la communication. Un canal ne peut connecter que deux machines.
- Si deux machines communiquent ensemble via des canaux, alors les signaux passent d'une machine à la file de l'autre machine à travers le canal de communication les reliant.

La capacité des canaux et des files associées aux MEFCs est théoriquement illimitée. En plus, un signal peut rester dans une file ou dans un canal pour une durée arbitraire avant d'être consommé. Pour définir formellement une MEFC, nous devons,

tout d'abord, définir la structure de la file. Une file est un triplet (A, N, C), où :

- A est un ensemble fini de l'alphabet de la file,
- N est un entier qui donne le nombre d'éléments dans la file,
- C est un sous-ensemble ordonné de A représentant le contenu de la file.

Les éléments de A et C sont distincts les uns des autres. Si plusieurs files sont associées à une MEFC alors leurs alphabets doivent être disjoints.

Une MEFC est formellement définie comme un quadruplé (S, s_0, M, δ), où :

- S est un ensemble fini d'états,
- $s_0 \in S$ est l'état initial,
- $M = \bigcup_{i=1}^{n} A_i \cup \varepsilon$ est l'ensemble de messages des files associées à la machine,
- $\delta : S \times M \longrightarrow S$ est la fonction des transitions. $\delta(s, m_{ij})$ est l'ensemble d'états que la machine peut atteindre à partir de l'état s après envoi (respectivement réception) de la sortie (respectivement l'entrée) m_{ij}.

La sémantique d'un système de MEFCs $(M_1, M_2, ..., M_n)$ est donnée par une machine décrivant les états globaux du système et les transitions entre ces états. Un état global consiste en une collection d'états locaux, un par MEFC, et le contenu de chaque canal. Plus formellement, un état global est un tuple $(s_1, s_2, ..., s_n, c_{2\to1}, c_{3\to1}, ..., c_{n\to1}, c_{1\to2}, c_{3\to2}, ..., c_{n\to1}, ..., c_{(n-1)\to n})$, où $s_i \in S_i$ et $c_{i\to j} \subseteq M_j$.

3.1.3 Les machines à états finis étendus

Les machines à états finis étendus (MEFEs)[CZ93, HLJ95, KC95, RDT95b, BDAR97, BDAR98] sont un modèle théorique utilisé pour la description des protocoles de communication. Contrairement aux MEFs, les MEFEs permettent de décrire en même temps l'aspect contrôle et l'aspect données des protocoles de communication.

Formellement, une MEFE est un quintuplet $E = (S, I, O, T, s_0)$, où :

- S est un ensemble fini non vide d'*états*,

- I est un ensemble fini non vide d'*interactions d'entrée* incluant l'interaction i_0 représentant l'absence d'interactions en entrée,
- O est un ensemble fini non vide d'*interactions de sortie* incluant l'interaction o_0 représentant l'absence d'interactions en sortie,
- $s_0 \in S$ est l'*état initial*,
- T est un ensemble fini non vide de *transitions*.

Chaque interaction d'entrée $i \in I$ est représentée par *?i(inlist)*, où i est le nom de l'interaction d'entrée et *inlist* est un ensemble de variables, appelées *paramètres*, appartenant à un ensemble V_I. Pour chaque variable $v \in V_I$, est associé un domaine fini $D(v)$. De même, chaque interaction de sortie $o \in O$ est représentée par *!o(outlist)*, où o est le nom de l'interaction de sortie et *outlist* est un sous-ensemble de variables appartenant à un ensemble V_O des *variables internes* du protocole. Pour chaque variable $v \in V_O$, est associé un domaine fini $D(v)$. L'union des ensembles V_I et V_O est l'ensemble V des variables de la MEFE ($V = V_I \cup V_O$).

Une transition de E est représentée par un 6−tuplet $t = (s, s', i, o, p, a)$, où $(s, s') \in S^2$ sont respectivement l'*état de départ* et l'*état d'arrivée* de la transition t, $i \in I$, $o \in O^*$, p est un *prédicat* appartenant à un ensemble fini P de prédicats sur les variables de la MEFE, a est appelée *la procédure de t* et consiste en un ensemble fini d'actions sur les variables de la MEFE.

Comme dans le cas des MEFs, on définit le domaine d'une MEFE E par l'ensemble des interactions d'entrée acceptables par la MEFE (c'est-à-dire, les triplets (s, i, p) tel que $(s, s', i, o, p, a) \in T$) et on le note D_E. Ainsi, on a $D_E \subseteq S \times I \times P$. Une MEFE E est dite *complètement spécifiée* si $D_E = S \times I \times P$; sinon elle est dite *partiellement spécifiée.*

La sémantique d'une MEFE E est donnée par l'ensemble de configurations de ses états. Une *configuration d'un état* $s \in S$ est un $(k+1)$-tuplet $(s, b_1, b_2, ..., b_k)$, où

$(b_1, b_2, ..., b_k) \in C$, où $C = \{(b_1, b_2, ..., b_k) \mid b_i \in D(v_i), v_i \in V\}$.

3.1.4 Le système de transitions étiquetées

Le système de transitions étiquetées (STE) [Tre92b, Bri88] est un formalisme utilisé pour la description des processus tel que la spécification, l'implantation et le test.

Formellement, un STE est un quadruplé $p = (S, L, T, s_0)$, 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 transitions,
- $s_0 \in S$ est l'état initial.

Les étiquettes appartenant à L représentent les *interactions observables* du système, alors que l'étiquette τ représente une *action interne non observable.* Une *trace* d'un STE est une séquence d'actions observable du système. L'ensemble des traces d'un STE I est noté par $Traces(I)$ et inclut la séquence vide ε. la Figure 3.1 donne quelques notations souvent utilisées dans l'étude des STEs.

Un système de transitions étiquetées est dit *non-déterministe* lorsqu'il ne permet pas de contrôler ses évolutions futures. On peut avoir deux sortes de non-déterminisme :

- Lorsqu'on a plusieurs transitions partant d'un même état et étiquetées par la même étiquette.
- Lorsqu'à partir d'un état, un choix est possible entre une interaction observable et une action interne (non observable).

Le modèle des STEs est à la base des systèmes communicants par le mécanisme de *rendez-vous.* Ce type de communication est modélisé par la composition parallèle

Notations	Définitions
$s \xrightarrow{\mu} s'$	$=_{def} (s, \mu, s') \in T$
$s \xrightarrow{\mu_1 \cdots \mu_n} s'$	$=_{def} \exists s_0, ..., s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} ... \xrightarrow{\mu_n} s_n = s'$
$s \xrightarrow{\mu_1 \cdots \mu_n}$	$=_{def} \exists s' : s \xrightarrow{\mu_1 \cdots \mu_n} s'$
$s \not\xrightarrow{\mu_1 \cdots \mu_n}$	$=_{def} not \exists s' : s \xrightarrow{\mu_1 \cdots \mu_n} s'$
$s \Longrightarrow s'$	$=_{def} s = s' or s \xrightarrow{\tau \cdots \tau} s'$
$s \xRightarrow{a} s'$	$=_{def} \exists s_1, s_2 : s \Longrightarrow s_1 \xrightarrow{a} s_2 \Longrightarrow s'$
$s \xRightarrow{a_1 \cdots a_n} s'$	$=_{def} \exists s_0, ..., s_n : s = s_0 \xRightarrow{a_1} s_1 \xRightarrow{a_2} ... \xRightarrow{a_n} s_n = s'$
$s \xRightarrow{\sigma}$	$=_{def} \exists s' : s \xRightarrow{\sigma} s'$
$s \not\xRightarrow{\sigma}$	$=_{def} not \exists s' : s \xRightarrow{\sigma} s'$
$init(p)$	$=_{def} \{a \in L \mid p \xRightarrow{a}\}$
$p \ after \ \sigma$	$=_{def} \{p' \mid p \xRightarrow{\sigma} p'\}$

TAB. 3.1: Quelques notations utilisées dans les STEs

des STEs. La composition parallèle de deux STEs $I_1 = (S_1, L_1, T_1, s_{01})$ et $I_2 = (S_2, L_2, T_2, s_{02})$, notée $I_1 \parallel I_2$, est un STE $I = (S, L, T, s_0)$ défini comme suit :

- $S \subseteq S_1 \times S_2$.
- $L = L_1 \cap L_2$.
- $s_0 = (s_{01}, s_{02})$.
- La relation de transitions, T, de I est obtenue au moyen des règles suivantes :
  - $(s_1, \tau, s_1') \in T_1 \Rightarrow \forall s_2 \in S_2, ((s_1, s_2), \tau, (s_1', s_2)) \in T$.
  - $(s_2, \tau, s_2') \in T_2 \Rightarrow \forall s_1 \in S_1, ((s_1, s_2), \tau, (s_1, s_2')) \in T$.
  - $a \in L_1 \cap L_2 \wedge (s_1, a, s_1') \in T_1 \wedge (s_2, a, s_2') \in T_2 \Rightarrow ((s_1, s_2), a, (s_1', s_2')) \in T$.

3.1.5 Les réseaux de Petri

Les réseaux de Petri[1] (voir par exemple [Rei85, Rei87, RT86, DCDMPS82]) sont un outil graphique et mathématique qui s'applique à un grand nombre de domaines où les notions d'événements et d'évolutions simultanées sont importantes. Parmi ces

[1] Carl Adam Petri est le père fondateur de ce formalisme. Il a présenté cette théorie en 1962 dans sa thèse intitulée *Communication avec des Automates* à l'université de Darmstadt.

domaines, on peut citer l'évaluation des performances des systèmes discrets, les protocoles de communication, la conception des systèmes distribués, les systèmes d'information, les interfaces homme-machine, etc...

La vue d'un système à base de réseaux de Petri se base sur les concepts d'événements et de conditions. Les événements sont les actions du système dont l'occurence est contrôlée par l'état du système. Les états du système, quant à eux, sont définis comme un ensemble de conditions. Une condition est une description logique de l'état du système. Les événements du système ne peuvent avoir lieu que si certaines conditions, dites *préconditions*, sont satisfaites. L'occurence des événements donne lieu à la satisfaction de certaines conditions dites *postconditions*.

Formellement, un réseau de Petri est un quadruplet $R = (P, T, Pre, Post)$, où :

- P est un ensemble fini de *places* représentant les conditions,
- T est un ensemble fini de *transitions* représentant les événements,
- $Pre : P \times T \longrightarrow N$ est l'application *incidence avant* (places précédentes),
- $Post : P \times T \longrightarrow N$ est l'application *incidence arrière* (places suivantes),

La sémantique d'un réseau de Petri est basée sur le *marquage* du réseau. Un réseau marqué est le couple $C = (R, M)$, où :

- R est un réseau de Petri,
- M est le marquage initial ; c'est une application $M : P \longrightarrow N$ associant à chaque place $p \in P$ un certain nombre de marques, appelées *jetons*.

L'exécution d'un réseau de Petri est contrôlée par le nombre et la distribution de jetons dans le réseau. Une transition t est *franchie* lorsque le nombre de jetons présents dans toutes ses places d'entrée est suffisant ; c'est-à-dire, $\forall p \in P : M(p) \geq Pre(p, t)$. Dans ce cas, les jetons sont enlevés des places d'entrée et d'autres nouveaux jetons seront crées et distribués sur les places de sortie de la transition.

Un réseau de Petri est généralement représenté par un graphe orienté avec deux type de noeuds : les places et les transitions. Une place est dessinée par un cercle et une transition par une barre. Un jeton, quant à lui, est représenté par un point. Un arc relie une place p à une transition t si et seulement si $Pre(p,t) \neq 0$. Un arc relie une transition t à une place p si et seulement si $Post(p,t) \neq 0$.

Les réseaux de Petri décrits dans cette section sont basés sur le principe de *causalité* entre les événements. Un événement a est la cause d'un événement b si et seulement si a précède toujours b. Dans ce cas, le temps est pris en considération d'une manière *qualitative*. Des efforts ont été fournis pour étendre les réseaux de Petri afin de prendre en considération le temps d'une manière *quantitative*. Par conséquent, nous avons vu apparaître les réseaux de Petri temporisés que nous décrirons un peu plus loin.

3.1.6 Les techniques de description formelle

Les techniques de description formelle (SDL, ESTELLE et LOTOS) sont basées sur les modèles des automates finis (MEFs, MEFEs et STEs) et ont été normalisées par l'ISO (International Standardization Organization) et l'ITU (International Telecommunication Union) pour spécifier, simuler, valider et générer automatiquement le code des systèmes distribués.

Le langage SDL

SDL (Specification and Description Language) [CCI93, Bel89] est un langage développé et normalisé par l'ITU. Il a été recommandé pour décrire formellement les systèmes de télécommunication afin d'analyser et d'interpréter correctement leurs comportements. Le développement du SDL a commencé en 1972 mais sa première version n'a été publiée qu'en 1976, suivie par de nouvelles versions en 1980, 1984, 1988, 1992, 1993 et 1996. En 1993, SDL a été étendu pour couvrir les concepts d'orienté-objets.

SDL offre deux formats syntaxiques pour la spécification d'un système : le format graphique (SDL/GR) et le format textuel (SDL/PR). Tous les deux sont des représentations équivalentes et dérivent de la même grammaire abstraite. Le comportement d'un système est formé de la combinaison de comportements des processus dans le système. Ces processus sont représentés par des MEFEs et s'exécutent en parallèle. La communication entre les processus se fait d'une manière asynchrone par des messages appelés *signaux*.

La description d'un système en SDL consiste à suivre une approche descendante dont le but est de surmonter la complexité du système. Cette méthodologie distingue trois niveaux d'abstraction :

- Niveau système : Au plus haut niveau d'abstraction, un système est vu comme un ensemble de blocs et un ensemble de canaux inter-blocs. Ces canaux constituent le seul moyen de communication entre les blocs et entre le système et son environnement. Chaque canal représente une communication unidirectionnelle. Ceci dit qu'une communication bidirectionnelle entre deux blocs donne lieu à deux canaux distincts.
- Niveau bloc : Chaque bloc du système contient un ensemble de processus et un ensemble de chemins des signaux inter-processus. Ces chemins constituent le seul moyen de communication entre les processus. En plus, ils peuvent être reliés aux canaux inter-blocs pour mettre en relation des processus appartenant à des blocs différents.
- Niveau processus : Le niveau processus décrit le comportement du système proprement dit. Un processus en SDL est une machine à états finis étendus manipulant des données stockées localement. Il a une adresse unique pour son identification et une file FIFO où les signaux arrivant sont stockés et d'où ils sont retirés.

Le langage Estelle

Estelle (Extended State Transition modEL) [ISO86, BD87] est un langage de description formelle développé par l'ISO pour la spécification des systèmes distribués et des systèmes concurrents. En particulier, Estelle est utilisé pour la description des protocoles de communication et des services. Le développement d'Estelle a commencé en 1981 mais sa standardisation n'a été faite qu'en 1988. Ceci dit qu'il est une technique de description formelle de deuxième génération. Estelle peut être vu comme une technique basée sur le modèle des MEFEs conjointement avec le langage Pascal. Un système modélisé en Estelle est une *structure hiérarchique* d'automates (composantes du système ou *modules*) qui :

- *s'exécutent en parallèle*, et
- *communiquent* entre eux par *échange de messages* et/ou par *un partage restreint* de certaines variables.

Pour spécifier un système en Estelle, on doit séparer la description des interfaces de communication entre les modules de la description interne de chaque module. L'interface de communication d'un module est définie à l'aide de trois concepts :

- Les points d'interactions qui sont les points d'accés au module via lesquels ce dernier échange des messages avec son environnement.
- Les canaux de communication qui sont associés aux points d'interaction et qui représentent le moyen de communication entre les modules.
- Les interactions qui sont les messages échangés entre le module et son environnement.

La description interne d'un module, quant à elle, est définie en terme d'une MEFE ; c'est-à-dire, par la définition de l'ensemble des états, l'état initial et la relation des transitions.

La structure hiérarchique d'une spécification écrite en Estelle est due au fait que la description d'un module peut inclure la description d'autres modules. En plus,

Estelle permet de créer des modules ayant le même comportement interne mais des interfaces de communication différentes. Ces modules sont appelés des *instances* du même module défini *textuellement* dans la spécification. Le nombre des instances d'un module peut changer *dynamiquement* puisque les instances peuvent être crées et détruites pendant l'exécution de la spécification.

Le langage LOTOS

LOTOS (Language of Temporal Ordering Specification) [ISO88, BB88] est un langage de description formelle qui a été recommandé par ISO pour décrire les systèmes distribués et plus particulièrement ceux liés à l'architecture OSI (Open System Interconnection). Le développement de LOTOS a duré de 1981 à 1986 et sa standardisation a été faite en 1988. Contrairement à ce que son nom l'indique, LOTOS n'est pas basé sur la logique temporelle mais plutôt sur l'*algèbre de processus* [Mil80a, Mil89a] et le type de données abstraits ACT ONE.

En LOTOS, un système est vu comme un *processus* consistant possiblement en plusieurs *sous-processus.* Par conséquent, une spécification LOTOS consiste en une *structure hiérarchique* de définitions de processus. Un processus est une entité capable d'exécuter des *actions internes non-observables* et d'*interagir* avec son environnement formé des autres processus. Cette interaction se fait par des *unités de synchronisation* appelées *événements* ou *actions* via des ports d'interactions.

À un certain niveau d'abstraction, un processus est représenté par une *boîte noire* avec des *portes de communication* via lesquelles le processus interagit avec son environnement. Ainsi, un processus est défini, en LOTOS, par son comportement constitué des séquences d'actions qui vont avoir lieu aux différentes portes du processus. Ce comportement est représenté par un STE.

Par ailleurs, la *synchronisation* entre deux processus P_1 et P_2 en LOTOS se fait

d'une manière à ce que P_1 *offre* un événement sur une porte et *se bloque* en attendant que P_2 répond par un autre événement. Ce mécanisme est appelé *synchronisation par rendez-vous.*

3.2 Modèles formels temporel de spécification

Dans la section précédente, nous avons passé en revue les différents modèles formels pour la spécification des aspects contrôle et données du comportement d'un système. Cependant, le développement des systèmes temps réel nécessitent de nouveaux modèles pour décrire leur aspect temporel.

3.2.1 Modèles du temps

Plusieurs modèles ont été proposés pour spécifier les systèmes temps réel. Selon la sémantique et le modèle du temps utilisé, ces modèles peuvent être classés en trois catégories [Dil90] :

- *Le modèle discret,*
- *Le modèle à horloge fictive*, et
- *Le modèle dense ou continu.*

Le modèle discret exige que la séquence du temps soit une série croissante des nombres entiers naturels. Ce modèle est approprié pour modéliser certains circuits numériques, où les valeurs d'un signal doivent changer précisément à l'arrivée du signal d'horloge. Un des avantages de ce modèle est qu'il est facilement transformable en une MEF. Chaque trace temporisée est transformée en une trace dans laquelle le temps progresse d'une unité à chaque étape. Ceci est fait par l'insertion d'un événement spécial, *NextTime*, modélisant le passage du temps de l'instant t à l'instant $t+1$. Une fois cette transformation est faite, on peut appliquer toutes les techniques connues sur les MEFs. Cependant, le modèle discret ne peut pas s'appliquer aux systèmes où les événements ne se produisent pas à des instants entiers. Dans de tel cas, le modèle discret approxime la continuité du temps par le choix à priori d'un *quantum* fixe pour

l'avancement du temps.

Le modèle à horloge fictive est similaire au modèle discret sauf que la séquence du temps ne doit être que croissante. L'interprétation de ce modèle est que les événements arrivent dans l'ordre spécifié par la séquence à des temps réels, mais seulement la partie entière de ces temps est enregistrée. Comme le modèle discret, le modèle à horloge fictive est facilement transformable en une MEF par l'insertion d'un nouvel événement, appelé *tick*, pour modéliser les *ticks* d'horloge. Une des limitations de ce modèle est que le temps n'est représenté que d'une manière approximative.

Le modèle dense, quant à lui, est un modèle général pour spécifier des systèmes temps réel. Dans ce modèle, le temps est dense et prend des valeurs dans l'ensemble des réels. Contrairement aux modèles précédents, le modèle dense est difficile à transformer en un modèle classique.

Dans ce qui suit, nous ne nous intéresserons qu'au modèle dense. Plus précisemment, nous présenterons les différents formalismes utilisés pour spécifier les systèmes temps réel à temps continu.

3.2.2 Les processus de temporisateurs parallèles

Un *Processus de Temporisateurs Parallèles* (PTP) [Čer92b] est un modèle formel pour spécifier les systèmes temps réel. Comme son nom l'indique, ce formalisme modélise le temps par un système de temporisateurs. Dans le modèle des PTPs, chaque temporisateur est un mécanisme abstrait représenté par une variable qui reçoit sa valeur initiale lors de l'exécution d'une transition. Une fois *activé*, la valeur d'un temporisateur décroit d'une manière synchrone par rapport à une horloge globale. Quand cette valeur atteint 0, le temporisateur est dit *expiré* et une transition peut être exécutée.

Formellement, le modèle des PTPs est défini à l'aide d'un graphe de transitions étiquetées et d'un automate de temporisateurs. Un graphe d'arcs (ou transitions) étiquetés G est un 6-tuple $(V, E, L, source, dest, lab)$, où :

- V est un ensemble fini de noeuds.
- E est un ensemble fini d'arcs.
- L est un ensemble fini d'étiquettes (les événements).
- $source : E \longrightarrow V$ est la fonction qui donne l'état de départ de chaque arc.
- $dest : E \longrightarrow V$ est la fonction qui donne l'état de destination de chaque arc.
- $lab : E \longrightarrow L$ est la fonction d'étiquetage des arcs.

Un automate de temporisateurs, quant à lui, est un quadruplet $\Phi = (G, T, \gamma, \phi)$ où :

- $G = (V, E, L, source, dest, lab)$ est un graphe d'arcs étiquetés,
- T est un ensemble fini de temporisateurs.
- $\gamma : E \longrightarrow 2^T$ est un sous-ensemble de temporisateurs conditionnant la transition e.
- $\phi(e) : T \longrightarrow T \cup Q^{\geq 0}$ est une fonction d'activation de temporisateurs. Chaque temporisateur peut être activé pour une durée rationnelle non négative, ou prendre la valeur d'un autre temporisateur, ou rester inchangé.

Chaque arc $e \in E$ dans le graphe G est coloré soit en rouge (exécution instantanée) soit en noir (possibilité d'attendre). L'ensemble des arcs de couleur rouge et celui des arcs de couleur noir sont notés par R et N respectivement.

L'ensemble des états d'un automate de temporisateurs $\Phi = (V, E, L, lab, T, \gamma, \phi)$ est donné par $S^{\Phi} = \{(v, \delta) \mid v \in V, \delta : T \longrightarrow Q^{\geq 0}\}$, où δ est une application qui associe à chaque temporisateur $t \in T$ une valeur rationnelle représentant sa valeur

courante.

En se basant sur le graphe de transitions étiquetées et l'automate de temporisateurs, on définit un Processus de Temporisateurs Parallèles comme un couple $P = (\Phi, s)$, où :

- $\Phi = (V, E, L, lab, T, \gamma, \phi)$ est un automate de temporisateurs.
- s est l'état initial du processus P.

Les étiquettes de transitions d'un automate de temporisateurs Φ sont des événements réels qui peuvent avoir lieu durant la vie des processus associés à Φ. Ces événements peuvent être l'envoi ou la réception d'un signal, la lecture ou l'écriture d'une donnée, un signal de synchronisation, ou toute autre sorte de communication avec l'environnement de l'automate. Pour toute action $\sigma \in L$, il existe certaines règles selon lesquelles un processus exécutant σ se transforme en un autre processus associé au même automate Φ (l'automate Φ change son état à l'exécution de l'action σ).

En plus des transitions sur les événements de l'ensemble L, tout processus $P = (\Phi, s)$ modélise l'écoulement du temps par des *transitions de délai*, $\varepsilon(d)$, où $d \in Q^{\geq 0}$. La relation $P \xrightarrow{\varepsilon(d)} Q$ signifie que le processus P se comportera comme le processus Q quand le temps progresse de d unités.

Formellement, les transitions d'un PTP sont données de la manière suivante : Soient $\Phi = (V, E, L, lab, T, \gamma, \phi)$ un automate de temporisateurs, $e \in E$, $\sigma \in L$ et $d \in Q^{\geq 0}$. Alors, on a :

- $\langle \Phi, (v, \delta) \rangle \xrightarrow{e:\sigma} \langle \Phi, (v', \delta') \rangle$ si les conditions suivantes sont satisfaites :
  - $e \in E$ tel que $source(e) = v$, $dest(e) = v'$ et $lab(e) = \sigma$.
  - Pour tout $t \in \gamma(e), \delta(t) = 0$; c'est-à-dire, la condition de la transition est satisfaite.

- Pour tout temporisateur $t \in T$, sa nouvelle valeur $\delta'(t)$ est donnée par l a fonction $\phi(e)$ de la manière suivante :
  - Si $\phi(e)(t) = c \in Q^{\geq 0}$, alors $\delta'(t) = c$.
  - Si $\phi(e)(t) = t' \in T$, alors $\delta'(t) = \delta(t')$.

- $\langle \Phi, (v, \delta) \rangle \xrightarrow{\varepsilon(d)} \langle \Phi, (v, \delta') \rangle$ si et seulement si :
  - Pour toute transition colorée en rouge $e \in R$ partante de v ($source(e) = v$), il existe $t \in \gamma(e)$ tel que $\delta(t) \geq d$; c'est-à-dire, aucune transition colorée en rouge ne peut s'exécuter durant ces d unités du temps. Si une transition colorée en rouge est tirable, aucun délai n'est possible au présent noeud du processus.
  - Pour tout $t \in T$, $\delta'(t) = \delta(t) \ominus d$, où $x \ominus y \stackrel{def}{=} max\{0, x - y\}$ pour tout x et y dans $\mathbf{R}$. Autrement dit, les valeurs de tous les temporisateurs décroissent vers 0 quant le temps progresse.

3.2.3 Le graphe des contraintes

Le Graphe des Contraintes (GC) [CL97] est un modèle formel qui a été proposé pour spécifier et tester des systèmes temps réel. Ce formalisme est basé sur la classification des contraintes temporelles faite par Taylor et Dasarathy [Tay80, Das85]. Ces derniers distinguent deux sortes de contraintes temporelles : *les contraintes de comportement* et *les contraintes de performance.* Les contraintes de comportement limitent la fréquence avec laquelle les entrées sont soumises à un système, tandis que les contraintes de performance spécifient la fréquence avec laquelle les sorties sont produites par un système.

Formellement, un graphe de contraintes est un graphe acyclique orienté $G = (V, E)$ avec un noeud spécial $S \in V$ représentant le noeud initial. L'ensemble des noeuds V est formé des noeuds étiquetés par des éléments de l'ensemble $D_C \cup \bar{D_C} \cup \{\varepsilon\}$. Les

termes D_C et $\bar{D_C}$ correspondent respectivement aux ensembles d'entrées et de sorties du système. ε représente toute action autre qu'un événement d'entrée ou de sortie.

Dans un *GC*, on distingue deux catégories d'arcs (ou transitions). La première catégorie regroupe tous les éléments appartenant à l'ensemble $V \times (D_T \times D_T) \times 2^{(D_C \cup \bar{D_C})} \times V$. Un élément de cet ensemble est noté par : $f \xrightarrow{[t_1,t_2)}_E g$. Les étiquettes $f, g \in V$ représentent respectivement la source et la destination de l'arc, $[t_1, t_2)$ est un intervalle de $(D_T \times D_T)$, et E est un ensemble possiblement vide des événements interdits durant l'intervalle $[t_1, t_2)$ $(E \subseteq 2^{(D_C \cup \bar{D_C})})$.

La deuxième catégorie d'arcs est un sous-ensemble de $V \times D_T \times 2^{(D_C \cup \bar{D_C})} \times V$. Un élément de cet ensemble est noté par : $f \xrightarrow{t}_E g$. Comme pour la première catégorie d'arcs, les étiquettes $f, g \in V$ représentent respectivement la source et la destination de l'arc, $t \in D_T$ est le délai exact entre f et g, et E est un ensemble possiblement vide des événements interdits entre f et g $(E \subseteq 2^{(D_C \cup \bar{D_C})})$.

3.2.4 Les réseaux de Petri temporisés

La première fois où les réseaux de Petri ont été étendus par des contraintes temporelles était en 1974 [Ram74]. Dès lors, plusieurs modèles de réseaux de Petri temporisés ont été proposés [Wal83, CR83, GMMP91]. Ces modèles diffèrent selon trois aspects importants :

- Le type de contraintes temporelles à exprimer.
- La localisation des contraintes temporelles.
- Le degré d'effet du temps sur le contrôle.

Les contraintes temporelles peuvent être décrites au moyen d'une seule valeur (*durée* ou *délai*) [CR83], ou un couple de valeurs (*intervalle de franchissement*) [Wal83], ou un ensemble de valeurs temporelles (*ensemble de franchissement*) [GMMP91].

Les contraintes temporelles peuvent être associées à trois différents emplacements à savoir *les places* [CR83], *les transitions* [GMMP91] et *les liens* [Wal83].

Le degré d'effet du temps sur le contrôle mène à des sémantiques différentes. Quand les contraintes temporelles ne peuvent pas forcer le franchissement d'une transition, on parle d'une *temporisation faible* [Wal83] ; sinon, on parle d'une *temporisation forte* [CR83].

Dans [CMS94], on trouve une comparaison entre ces différents modèles et une définition générale d'un réseau de Petri temporisé. Cette définition inclut tous les emplacements où les contraintes temporelles peuvent être utilisées, donnant lieu ainsi à un nouveau type de réseaux de Petri temporisés, appelés *réseaux de Petri temporisés généraux*. Soient Σ un ensemble fini d'alphabets et $D^\infty = D \cup \{\infty\}$ un domaine étendu du temps ($D \subseteq \mathbf{R}^{\geq \mathbf{0}}$). Un Réseau de Petri Temporisé Général (RPTG) défini sur Σ, D est un 7-tuple $N = (S, T, L, l, \delta, \Delta, M)$, où :

- S est un ensemble fini de places.
- T est un ensemble fini de transitions tel que $S \cap T = \emptyset$.
- $L = I \cup O$ est un ensemble fini de liens, où $I \subseteq S \times T$ est un ensemble de liens entrants, et $O \subseteq T \times S$ est un ensemble de liens sortants.
- $l : T \longrightarrow \Sigma \cup \{\varepsilon\}$ est la fonction d'étiquetage des transitions.
- δ et Δ sont respectivement les fonctions de bornes inférieure et supérieure du temps tel que l'une des conditions suivantes est satisfaite :
  - $\delta : S \longrightarrow D$ et $\Delta : S \longrightarrow D^\infty$.
  - $\delta : T \longrightarrow D$ et $\Delta : T \longrightarrow D^\infty$.
  - $\delta : I \longrightarrow D$ et $\Delta : I \longrightarrow D^\infty$.
- $M : S \longrightarrow \mathbf{N}$ est le marquage initial.

3.2.5 Les automates temporisés

Les Automates Temporisés (ATs) [AD94, NSY92, DY95] sont un formalisme adéquat pour la description du comportement d'un système temps réel. Ils sont une extension des ω-automates (automates de Büchi et automates de Muller) [Bô2, Cho74] par l'ajout de variables d'horloges et de contraintes d'horloges pour conditionner l'exécution des transitions.

Formellement, un AT est un quintuplet $A = (\Sigma_A, L_A, l_A^0, C_A, T_A)$, où :

- Σ_A est un ensemble fini d'alphabets (événements).
- L_A est un ensemble fini d'emplacements.
- $l_A^0 \in L_A$ est l'emplacement initial.
- C_A est un ensemble fini d'horloges initialisées toutes à zéro en l_A^0.
- $T_A \subseteq L_A \times L_A \times \Sigma_A \times 2^{C_A} \times \Phi(C_A)$ est l'ensemble de transitions.

Un tuple $(l, l', a, \lambda, G) \in T_A$, noté par $l \xrightarrow{a,\lambda,G}_A l'$, représente une transition de l'emplacement l à l'emplacement l' sur l'événement a. Le sous-ensemble $\lambda \subseteq C_A$ donne les horloges à remettre à zéro suite à cette transition, et $G \in \Phi(C_A)$ est une contrainte temporelle sur l'exécution de la transition. Le terme $\Phi(C_A)$ représente l'ensemble des contraintes sur C_A, formé à partir des formules atomiques de la forme $x\ op\ m$, où $x \in C_A$, $op \in \{<, \leq, =, >, \geq\}$ et $m \in N$. Le choix des entiers naturels comme bornes dans les contraintes est cruciale puisqu'il permet de discrétiser l'ensemble des réels en des intervalles entiers réduisant ainsi l'espace d'états du système.

Chaque horloge $x \in C_A$ prend ses valeurs dans $\mathbf{R}$, croit de manière synchrone par rapport à une horloge globale et mesure le temps écoulé depuis sa dernière réinitialisation.

Pour définir le modèle sémantique d'un AT, on définit, tout d'abord, la notion d'*interprétation d'horloges* et la notion d'*état*. Une interprétation d'horloges sur un

ensemble d'horloges C est un "mapping" qui associe à chaque horloge $x \in C$ une valeur dans $\mathbf{R}^{\geq \mathbf{0}} \cup \{+\infty\}$. L'ensemble des interprétations d'horloges est noté par $V(C)$. Une interprétation d'horloges v satisfait une contrainte temporelle G, et on le note $v \models G$, si et seulement si G est évaluée à vraie sous v.

Pour tout $d \in \mathbf{R}^{\geq \mathbf{0}}$, $v + d$ est une interprétation d'horloges qui affecte la valeur $v(x) + d$ à toute horloge x. De même, pour tout $X \subseteq C$, $[X := d]v$ est une interprétation d'horloges qui assigne la valeur d à toute horloge $x \in X$ et ne modifie pas les valeurs des autres horloges. Intuitivement, l'interprétation d'horloges $v + d$ représente les valeurs des horloges quand le temps s'écoule de d unités. Par contre, l'interprétation d'horloges $[X := d]v$ est utilisée pour déterminer les valeurs d'horloges quand le système exécute une transition explicite.

Un état d'un *AT* A est un couple (l, v), où $l \in L_A$ et $v \in V(C_A)$. L'état initial de A est le couple (l_A^0, v_0), où l_A^0 est l'emplacement initial de A et v_0 est une interprétation d'horloges tel que $v_0(x) = 0$ pour toute horloge $x \in C_A$. L'ensemble des états de A est noté par S_A.

Nous distinguons deux types de transitions dans un *AT* :

- Les *transitions de délai* : l'automate change son état à chaque écoulement du temps. Les transitions de ce type ont la propriété d'additivité de Wang [Yi90] : $s \xrightarrow{\varepsilon(d+d')} s' \Leftrightarrow \exists s'' \in S_A : s \xrightarrow{\varepsilon(d)} s'' \wedge s'' \xrightarrow{\varepsilon(d')} s'$ (ε signifie pas d'action).
- Les *transitions explicites* : à partir d'un état donné (l, v), l'automate exécute plusieurs transitions de délai et quand il atteint un état (l, v') où la contrainte temporelle d'une transition sortante de l est satisfaite, l'automate peut exécuter cette transition. De telles transitions sont notées par $s \xrightarrow{a} s'$, où $a \in \Sigma_A$.

Le modèle sémantique d'un *AT* A est donné par un système de transitions étiquetées temporisé $S_t(A) = (S_A, \Sigma_A \cup \mathbf{R}^{\geq \mathbf{0}}, \longrightarrow, (l_A^0, v_0))$ dont l'ensemble d'états est S_A, l'en-

semble d'alphabets est l'union de l'alphabet de A (Σ_A) et l'ensemble de délais ($\mathbf{R}^{\geq \mathbf{0}}$), et la relation de transitions inclut les deux types de transitions cités précédemment. Cependant, ce système de transitions étiquetées est infini à cause de l'infinité des transitions de délai ou de l'espace d'états. Pour le réduire, nous verrons au chapitre 5 comment mettre ensemble tous les états équivalents (ayant le même comportement) pour construire ce qu'on appelle le graphe des régions.

3.2.6 Les machines temporisées communicantes

Un système temps réel est, dans la plus part des cas, composé de plusieurs processus s'exécutant d'une manière concurrente et communiquant entre eux sous des contraintes temporelles. Pour spécifier de tels systèmes, on a développé les machines temporisées communicantes (MTCs) [Kan95]. Dans ce formalisme, chaque processus du système, dit aussi processus MTC, est décrit par une machine à états finis avec des variables. Ces dernières sont de deux types : les variables de données et les variables d'horloges. Les variables des données sont les paramètres des messages échangés entre les processus pour contrôler leurs exécutions. Cependant, les variables d'horloges servent à spécifier l'aspect temporel du processus.

Formellement, une machine temporisée communicante M est un tuple (L, l_0, V, I, C, T), où :

- L est un ensemble fini d'emplacements.
- l_0 est l'emplacement initial.
- V est un ensemble fini de variables de données et d'horloges.
- I est la fonction d'initialisation des variables de données.
- C est un ensemble fini de canaux de communication.
- T est la relation des transitions. Chaque élément de T est de la forme (l_1, c, a, h, l_2), où l_1 et l_2 sont des emplacements, c est une condition sur les variables, a est

une action, et h est un ensemble d'affectations.

L'ensemble des variables de données et l'ensemble des variables d'horloges sont disjoints. La communication entre les processus se fait d'une manière synchrone à travers les canaux de communication. En effet, toutes les MTCs connectées à un même canal doivent s'engager simultanément dans la communication. La réception d'un message e paramétré par l'ensemble des variables X est notée par $e?X$. Cependant, l'envoi d'un message e paramétré par une expression exp est noté par $e!(exp)$. D'un autre côté, l'ensemble h est subdivisé en deux parties : les affectations concernant les variables de données h_d et celles portant sur les variables d'horloges h_c.

Pour décrire le comportement d'un système temps réel par une seule MTC, on compose les MTCs de tous les processus de ce système.

La sémantique d'un processus MTC est similaire à celle d'un automate temporisé. En effet, le processus commence à son emplacement initial l_0 en mettant les variables d'horloges à zéro et les autres variables à des valeurs arbitraires. Les valeurs des horloges croissent uniformément avec le temps, mais les variables de données restent inchangées. À tout instant, le processus peut exécuter une transition si la condition correspondante est satisfaite par les valeurs de différentes variables. Si la transition est tirée, le processus modifie éventuellement les valeurs de ses variables et/ou change son emplacement.

3.3 Conclusion

Dans ce chapitre, nous avons présenté les modèles formels les plus utilisés pour spécifier les systèmes informatiques. Ces modèles ont des expressivités différentes selon les aspects des systèmes à décrire. C'est selon leur expressivité temporelle que nous avons classé ces modèles en deux catégories : les modèles non temporels de spécification et les modèles temporels de spécification. Les premiers ne permettent

pas de décrire suffisamment l'aspect temporel d'un système alors que les derniers permettent de le faire.

Comme la spécification formelle est la première phase du cycle de développement des systèmes informatiques, elle a une influence majeure sur le processus de vérification et du test. En effet, la génération de tests est plus ou moins complexe selon le modèle formel de spécification et les propriétés qu'on veut tester. Autrement dit, la génération de tests dépend du modèle formel utilisé comme spécification de référence de l'implantation qu'on veut tester.

Dans le chapitre suivant, nous allons présenter les méthodes de génération de tests à partir des modèles formels les plus utilisés que ce soient temporels ou non. Certaines de ces méthodes seront réutilisées plus tard soit pour la présentation de nos contributions soit pour la comparaison avec nos méthodes.

Chapitre 4

Génération de tests : état de l'art

La génération de tests est la première étape dans le processus du test des logiciels. Elle consiste à dériver systématiquement des cas de tests à partir de la spécification de réference de l'implantation qu'on veut tester.

Dans ce chapitre, nous présentons l'état de l'art de la génération de tests à partir des modèles formels de spécification. Nous nous intéressons surtout aux modèles et aux techniques qui sont en relation avec notre travail ou que nous allons utiliser dans les prochains chapitres. Plus spécifiquement, nous présentons le test basé sur le modèle des MEFs, le test fondé sur les MEFEs et le test temporisé.

4.1 Le test basé sur le modèle des MEFs

Le modèle des machines à états finis est la base de la plupart des activités de test. Il a été utilisé depuis les années 50s dans le test des circuits séquentiels (matériel), des protocoles de communication et des systèmes orientés objets [Yao95]. Le test basé sur le modèle des MEFs peut être formalisé comme un problème de relation entre deux MEFs : la première, M_I, modélise l'implantation à tester et la deuxième, M_S, décrit la spécification de réference. Le test consiste à vérifier si M_I est une implantation valide de M_S. Pour ce faire, on définit une relation de conformité (voir section 7.1) pour

relier M_I à M_S. La relation de conformité la plus utilisée est l'*équivalence de traces* (voir section 3.1.1). Elle se base sur le comportement observable de l'implantation à la suite de l'application d'une suite de test.

La génération de tests à partir d'une MEF est souvent faite sous certaines hypothèses sur la spécification M_S, l'implantation M_I et les types de fautes (le modèle de fautes) qui peuvent être présentes dans l'implantation M_I. Les hypothèses sur la spécification sont utilisées pour s'assurer que seules les spécifications ayant certaines propriétés désirables sont traitées. Les hypothèses sur l'implantation et celles concernant les types de fautes possibles sont utilisées pour limiter l'univers des implantations à considérer. Sans ces hypothèses, toute MEF peut être considérée comme une implantation possible de la spécification et donc le nombre des implantations à considérer sera infini.

La spécification est généralement supposée être une MEF :

- Déterministe.
- Réduite.
- Fortement connexe.
- Complètement spécifiée.

Par contre, l'implantation est souvent supposée être une MEF :

- Ayant un certain nombre maximal d'états.
- Ayant un ensemble d'alphabet connu.
- Fortement connexe.
- Produisant une réponse à une entrée en un temps fini.
- Implantant correctement une action de remise à son état initial.

Avec ces hypothèses, le modèle de fautes des MEFs peut se résumer en quatres types de fautes [BDD+91] :

- Fautes de sorties : la sortie d'une transition est différente de celle attendue.
- Fautes de transfert : l'état d'arrivée d'une transition est erroné.
- Fautes de transfert avec états additionnels : l'état d'arrivée d'une transition est un nouvel état. Certains types de fautes ne peuvent être modélisés que par des états additionnels jumelés avec des fautes de transfert aboutissant à ces états.
- Fautes de transitions additionnelles ou manquantes : On suppose souvent que l'automate est déterministe et complètement spécifiée; c'est-à-dire, que pour chaque paire d'état et d'entrée, il y a exactement une transition correspondante. L'absence de cette transition est une faute de ce type.

Plusieurs méthodes de génération de tests à partir d'une MEF ont été développées. Il en existe deux catégories : les méthodes fondées sur une sorte d'*identification des états* et celles basées sur un *parcours simple de toutes les transitions.* Dans ce qui suit, nous présentons une synthèse de toutes ces méthodes [CA97, DSA+99].

4.1.1 La méthode du tour de transitions

La Méthode du Tour de Transitions [NT81] consiste à générer une seule séquence de transitions, dite *Tour de Transitions*, dans laquelle chaque transition de la MEF figure au moins une fois. Bien évidemment, la séquence générée peut contenir des entrées redandantes générant des cycles dans la séquence. Ceci peut être évité par l'optimisation du tour de transitions. Des algorithmes efficaces pour la dérivation du tour de transitions pour certaines classes de MEFs ont été étudiés dans[EJ73].

Le problème de trouver un tour de transitions optimal dans une MEF est un problème connu dans la théorie des graphes. En effet, une MEF est vue comme un graphe orienté et le tour de transitions optimale ressemble exactement à un *Tour d'Euler.* Un tour d'Euler dans un graphe orienté est une séquence de transitions qui commence et se termine au même état et qui contient chaque transition une et une seule fois. Pour qu'un tour d'Euler existe dans un graphe orienté, il suffit que le graphe soit :

- Fortement connexe.
- Symmétrique.

Un graphe est dit symmétrique si et seulement si le nombre de transitions entrantes à chaque état du graphe est égal au nombre de transitions sortantes du même état.

Un graphe non-symmétrique peut être transformé en un autre symmétrique par une procédure d'*augmentation* [Nah95]. Ceci consiste à dupliquer certaines transitions jusqu'à ce que le graphe devienne symmétrique.

La couverture de fautes d'une suite de test générée par la méthode du tour de transitions n'est pas complète. En effet, la suite de test ne vérifie que les sorties des transitions. Ceci n'est pas suffisant puisqu'il faut aussi vérifier l'état où se trouve l'IST après chaque transition.

Dans l'exemple de la Figure 4.1, un tour de transitions possible est formé de la séquence de transitions $t_1.t_4.t_3.t_6.t_7.t_8.t_2$. Il est clair que l'état d'arrivée de la dernière transition, t_2, n'est pas vérifié. De ce fait, si la transition t_8 mène à s_2 au lieu de s_1, cette erreur ne sera pas détectée.

4.1.2 La méthode de séquence de distinction

La Méthode de Séquence de Distinction [Gon70] est applicable si la MEF a une *séquence de distinction (DS)*. Une séquence de distinction est une séquence d'entrées qui produit une séquence de sorties différente pour chaque état de la MEF.

La méthode DS consiste en deux étapes : la *vérification des états* et la *vérification des transitions*. Durant la première étape, l'implantation doit donner la bonne réponse à la séquence de distinction pour chaque état. La deuxième étape, quant à elle, consiste

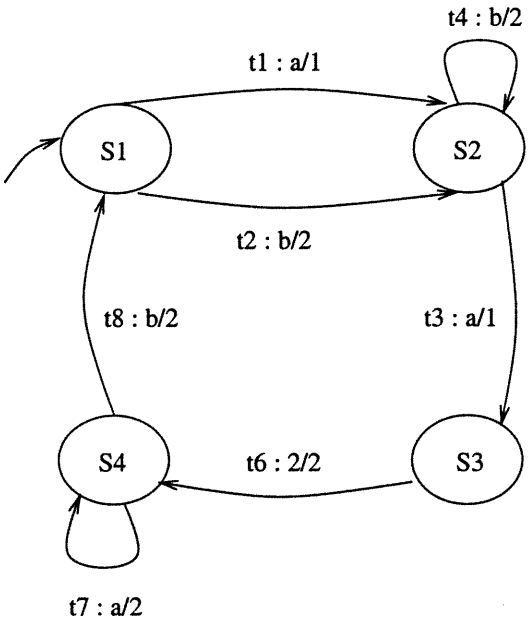

FIG. 4.1: Un exemple de MEF

à vérifier chaque transition de la MEF.

La vérification des états est faite par l'application de la procédure suivante à chaque état s_i de la MEF :

- L'implantation est amenée à un état i_i, supposé isomorphe à l'état s_i, par l'application du message de remise à l'état initial suivi par un *préambule* de l'état initial s_0 à l'état s_i.
- La séquence DS est soumise à l'implantation pour voir si sa réponse est vraiment égale à la réponse attendue.

Cette phase assure que l'implantation a au moins n états $i_1, i_2, ..., i_n$ qui sont accessibles par les mêmes préambules que leurs états isomorphes $s_1, s_2, ..., s_n$ dans la spécification.

Cependant, plusieurs transitions sont utilisées dans les préambules et la séquence de distinction durant la vérification des états. Ces transitions peuvent ne pas être

correctement implantées. C'est l'étape de vérification des transitions qui a le rôle de tester chaque transition $s_i \xrightarrow{i/o} s_j$ de la manière suivante :

- L'implantation est amenée à l'état i_i, isomorphe à s_i, par l'application du message de remise à l'état initial suivi par un préambule *Préambule*(s_i) ;
- L'entrée i est soumise à l'implantation et la réponse est observée pour vérifier que c'est bien o ;
- Par l'application de la séquence DS, le nouvel état de l'implantation est testé pour vérifier si c'est bien i_j.

La méthode DS garantit une couverture complète de fautes. Ceci dit qu'aucune implantation fautive avec le même nombre d'états que la spécification ne peut passer avec succés les cas de test générés par la méthode DS. La Figure 4.2 et la Table 4.1 montrent respectivement une MEF possédant une $DS = a.a$ et les séquences de sorties ainsi que les préambules correspondant à chacun de ses états.

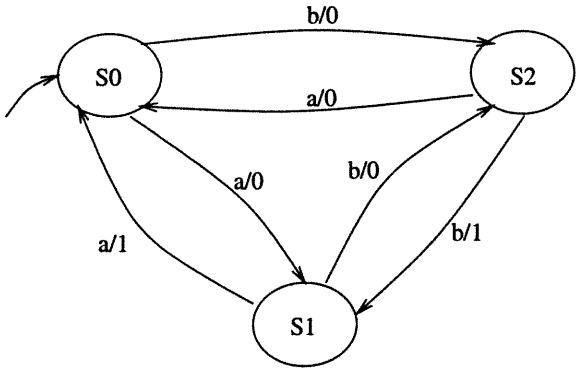

FIG. 4.2: Un exemple de MEF possédant DS

État	Sortie pour $DS = a.a$	Préambule(s_i)
s_0	0.1	Nul
s_1	1.0	$a/0$
s_2	0.0	$b/0$

TAB. 4.1: Une séquence DS pour la MEF de la Figure 4.2

4.1.3 La méthode W

La méthode DS a l'avantage d'avoir une couverture complète de fautes. Cependant, son inconvénient majeur est que la séquence DS n'existe pas pour toutes les MEFs. Ceci rend l'application de la méthode DS très limitée et a constitué en quelque sorte une motivation pour la recherche d'une autre méthode plus générale. La méthode W [Cho78] est une méthode générale applicable à toutes les MEFs. Elle se base sur le fait que pour toutes les MEFs, il existe un ensemble fini de séquences d'entrées, dit *l'ensemble de caractérisation W*, tel que les séquences de sorties sont uniques pour chaque état.

En plus de l'ensemble W, la méthode W consiste à choisir un deuxième ensemble de séquences d'entrées, dit *l'ensemble de couverture des transitions P*; c'est-à-dire, un ensemble de séquences d'entrées qui vérifie : pour chaque transition $s_i \xrightarrow{i/o} s_j$ de la spécification, il existe deux séquences d'entrées p et $p.i$ dans P tel que p amène l'automate de la spécification de son état initial à s_i.

Pour appliquer la méthode W, la spécification doit être *réduite* afin de garantir l'existence de l'ensemble W. De même, pour garantir une couverture complète de fautes des cas de test générés par la méthode W, l'implantation et la spécification doivent avoir le même alphabet d'entrées et doivent être complètement spécifiées et déterministes. En plus, tous les états de l'implantation et de la spécification doivent être accessibles à partir de l'état initial. L'implantation doit contenir une opération de remise à l'état initial correctement implantée. Enfin, le nombre d'états dans l'implantation ne doit pas dépasser un entier m plus grand ou égal au nombre d'états dans la spécification.

La méthode W produit la suite de test formée de la concaténation des ensembles Z et P (c'est-à-dire, $P.Z$), où $Z = (\{\varepsilon\} \cup I \cup I^2 \cup ... \cup I^{m-n}).W$. L'utilisation de l'ensemble Z au lieu de l'ensemble W est due principalement au fait que le nombre

d'états, m, dans l'implantation peut être plus grand au nombre d'états, n, dans la spécification. Si $m = n$, nous aurons alors $Z = W$ et par conséquent, la suite de test sera $P.W$.

La suite de test $P.Z$ détecte toutes les erreurs de sorties et de transfert dans une IST tant que le nombre d'états dans cette implantation ne dépasse pas m. La Figure 4.3 et la Table 4.2 montrent respectivement une MEF réduite et l'ensemble W correspondante.

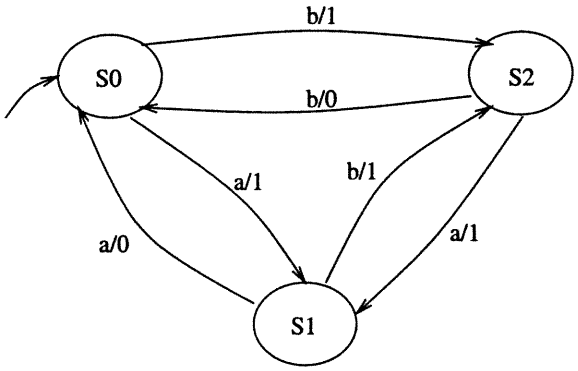

FIG. 4.3: Un exemple de MEF réduite

État	Sortie pour $W = \{a, b\}$	Préambule(s_i)
s_0	$\{1, 1\}$	Nul
s_1	$\{0, 1\}$	$a/1$
s_2	$\{1, 0\}$	$b/1$

TAB. 4.2: Un ensemble W pour la MEF de la Figure 4.3

4.1.4 La méthode UIO

La méthode de *séquences uniques d'entrée/sortie* ou *la méthode UIO* [SD88a] consiste à générer des séquences de tests qui vérifient si l'état d'arrivée de chaque transition est le bon (ou le transfert est bon). Pour y parvenir, elle utilise la séquence unique d'entrée/sortie de l'état d'arrivée. Une séquence unique d'entrée/sortie d'un

état permet de distinguer cet état de tous les autres états.

Pour appliquer la méthode UIO, la MEF de la spécification doit contenir une opération de remise à l'état initial. De même, chaque état de cette MEF doit avoir sa propre séquence d'entrée/sortie. La procédure de test est la suivante : Pour chaque transition $s_i \xrightarrow{i/o} s_j$,

- Ramener la MEF à son état initial, s_0, moyennant l'opération de remise à l'état initial.
- Trouver le chemin le plus court de l'état initial à l'état s_i.
- Appliquer l'entrée i pour permettre l'exécution de la transition et vérifier si la sortie est bien o.
- Appliquer la séquence unique d'entrée/sortie de l'état s_j pour vérifier si c'est le bon.

Il arrive parfois qu'une séquence unique d'entrée/sortie n'existe pas pour un état. Dans ce cas, la technique de signature est utilisée. La signature d'un état s_i est une séquence formée d'un ensemble de séquences minimales d'entrées/sorties, $IO(s_i, s_j)s$, chacune d'elles commence à s_i et distingue s_i d'un autre état s_j $(j \neq i)$.

Dans l'article originale où la méthode UIO a été proposée, les auteurs se contentaient d'utiliser uniquement la vérification des transitions (voir ci-dessus) pour générer la suite de test parce qu'ils croyaient que cela suffit pour garantir une bonne couverture de fautes. Dans [VCI89], Vuong s'est rendu compte de cette lacune en montrant par des exemples que certaines erreurs peuvent s'échapper aux testeurs même si l'implantation passe avec succès les cas de test générés par la méthode UIO. Pour cela, il a recommendé d'ajouter une étape de vérification des états à la méthode UIO pour donner lieu à une nouvelle méthode, dite *la méthode UIOv.* Cette étape de vérification des états consiste à s'assurer que les séquences uniques d'entrée/sortie sont aussi uniques dans l'IST. Pour ce faire, il faut vérifier que pour chaque état s_i de la spécification, la séquence $UIO(s_i)$ est acceptée par son état isomorphe dans l'implantation et elle

est *rejetée* par tous les autres états. La procédure du rejet d'une séquence $UIO(s_i)$ par un état s_j est la suivante :

- Ramener l'implantation à un état i_j, supposé isomorphe à l'état s_j, par l'application du message de remise à l'état initial suivi par un préambule de l'état initial s_0 à l'état s_j.
- Soumettre la séquence $UIO(s_i)$ à l'implantation et vérifier que la réponse observée est vraiment *différente* de celle obtenue par l'application de $UIO(s_i)$ à l'état s_i.

Dans la Figure 4.4 et la Table 4.3, nous présentons respectivement une MEF et les UIOs correspondantes à chacun de ses états.

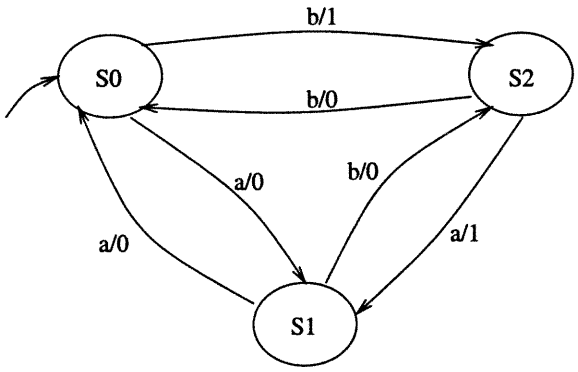

FIG. 4.4: Un exemple de MEF avec UIOs

État	$UIO(s_i)$	Préambule(s_i)
s_0	$b/1$	Nul
s_1	$a/0.b/1$	$a/1$
s_2	$a/1$	$b/1$

TAB. 4.3: Les séquences UIOs pour la MEF de la Figure 4.4

4.1.5 La méthode Wp

La méthode Wp [FBK+91] est une version optimisée et raffinée de la méthode W dont le but est de réduire la longueur de la suite de test. Au lieu d'utiliser tout l'en-

semble de caractérisation W pour vérifier l'état d'arrivée des transitions, seulement certains sous-ensembles de W sont généralement utilisés. Ces sous-ensembles sont dits *ensembles d'identification des états.* Un ensemble $W_i \subseteq W$ est dit ensemble d'identification de l'état s_i si et seulement si W_i est minimal et pour chaque état $s_j (j \neq i)$, il existe une séquence d'entrée appartenant à W_i qui génère deux comportements différents dans s_i et s_j. L'union de tous les W_i forme l'ensemble de caractérisation W.

La génération des cas de test par la méthode Wp consiste en deux phases :

- La première phase vérifie que tous les états de la spécification sont identifiables dans l'implantation, et que chaque état i_i dans l'implantation peut être identifié par l'ensemble W_i. En même temps, les sorties et les états d'arrivée de toutes les transitions utilisées dans les préambules sont testées. Les cas de test permettant d'atteindre cet objectif sont données par $Q.(\{\varepsilon\} \cup I \cup I^2 \cup ... \cup I^{m-n}).W$, où W est l'ensemble de caractérisation, m est le nombre maximal d'états de l'implantation, et Q est l'ensemble de *couverture d'états.* L'ensemble Q est formé de séquences d'entrées permettant d'atteindre tous les états de la spécification à partir de son état initial.
- La deuxième phase vérifie l'implantation de toutes les transitions de la spécification qui n'ont pas été testées durant la première phase. La suite de test générée par cette phase est donnée par l'ensemble des cas de test suivants : $R.I[m-n] \otimes \{W_0, W_1, ..., W_{n-1}\} = \bigcup_{i_1 \in R}\{i_1\}.(\bigcup_{i_2 \in I[m-n]}\{i_2\}.W_j)$, où $I[m-n] = (\{\varepsilon\} \cup I \cup I^2 \cup ... \cup I^{m-n})$, R est l'ensemble de toutes les transitions de la spécification ne faisant pas partie de Q, W_j est l'ensemble d'identification de l'état s_j, s_j est l'état atteint de s_i par la séquence d'entrées i_2, et s_i est l'état atteint à partir de l'état initial s_0 par la séquence d'entrées i_1.

Enfin, les hypothèses de test de la méthode Wp sont les mêmes que celles de la méthode W et sa couverture de fautes est complète. La Table 4.4 montre les ensembles W et $W_i (i = 0, 1, 2)$ de la MEF de la Figure 4.3 ainsi que les sorties correspondantes.

État	Sortie pour $W = \{a, b\}$	Préambule(s_i)	Ensemble W_i	Sortie pour W_i
s_0	$\{1, 1\}$	*Nul*	$\{a, b\}$	$\{1, 1\}$
s_1	$\{0, 1\}$	$a/1$	$\{a\}$	$\{0\}$
s_2	$\{1, 0\}$	$b/1$	$\{b\}$	$\{0\}$

TAB. 4.4: Les ensembles W et W_i pour la MEF de la Figure 4.3

4.1.6 La méthode Wp généralisée

La *Méthode Wp Généralisée* [LBP94] est une extension de la méthode Wp [FBK+91] pour générer des cas de test à partir d'une MEF *nondéterministe observable (MEFNO)*. Elle utilise les mêmes ensembles que la méthode Wp mais avec l'hypothèse de test d'*équité* (voir Section 2.2.3) sur l'implantation. Cette hypothèse veut dire qu'il est possible d'atteindre tous les états accessibles par une séquence d'entrées t dans une implantation en appliquant t à celle-ci un nombre fini de fois. Sans cette hypothèse, aucune suite de test ne peut garantir une couverture complète de fautes pour une implantation nondéterministe.

La Méthode Wp Généralisée est applicable aux systèmes concurrents modélisés par un ensemble de MEFs communicant entre elles via des files FIFOs et des canaux. Dans ce cas, la démarche est la suivante. Premièrement, les MEFs du système sont composées, en utilisant l'analyse d'accessibilité, pour obtenir une seule MEFNO. Ensuite, la méthode Wp généralisée est appliquée sur la MEFNO résultat.

La Figure 4.5 et la Table 4.5 montrent respectivement une MEFNO et les ensembles W et $W_i (i = 0, 1, 2, 3)$ correspondants.

4.1.7 La méthode W harmonisée

La *méthode W harmonisée* [Pet91] est utilisée pour générer des cas de test à partir d'une MEF déterministe et partiellement spécifiée. Formellement, un tuple $(H_1, H_2, ..., H_n)$ est dit une *identification harmonisée d'états* d'une MEF M si et seule-

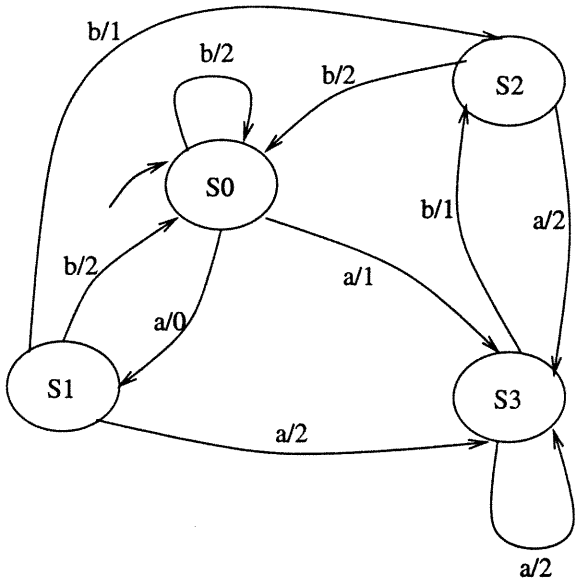

FIG. 4.5: Un exemple de MEFNO

État	Sortie pour $W = \{a, b\}$	Préambule(s_i)	Ensemble W_i	Sortie pour W_i
s_0	$\{0, 1, 2\}$	Nul	$\{a\}$	$\{0, 1\}$
s_1	$\{2, 1\}$	$a/0$	$\{b\}$	$\{1, 2\}$
s_2	$\{2\}$	$a/0.b/1$	$\{a, b\}$	$\{2\}$
s_3	$\{1, 2\}$	$a/1$	$\{b\}$	$\{1\}$

TAB. 4.5: Les ensembles W et W_i pour la MEFNO de la Figure 4.5

ment s'il satisfait les conditions suivantes :

- Pour tout $i = 1, 2, ..., n$, H_i est un ensemble de séquences d'entrées acceptables à l'état s_i de la MEF M.
- Pour tout $i \neq j$, il existe des séquences d'entrées $\alpha \in H_i$ et $\beta \in H_j$ commençant par une sous-séquence commune donnant deux séquences de sorties différentes en s_i et s_j.

Il est à noter que toute MEF *réduite* a une identification harmonisée d'états. En effet, pour toute paire d'états (s_i, s_j) d'une MEF réduite M, il existe une séquence d'entrées α tel que α est une séquence acceptable en s_i et s_j et donnant deux séquences de sorties différentes en ces états. Ainsi, la séquence α peut être incluse dans les

séquences H_i et H_j correspondant respectivement aux états s_i et s_j. La répétition de la même procédure pour toutes les paires d'états de la machine donne lieu à l'identification harmonisée d'états $(H_1, H_2, ..., H_n)$ de M.

La méthode W harmonisée garantit une couverture complète de fautes.

La Figure 4.6 et la Table 4.6 montrent respectivement une MEF et l'ensemble des identifications harmonisées $H_i(i = 0, 1, 2, 3)$ de ses états.

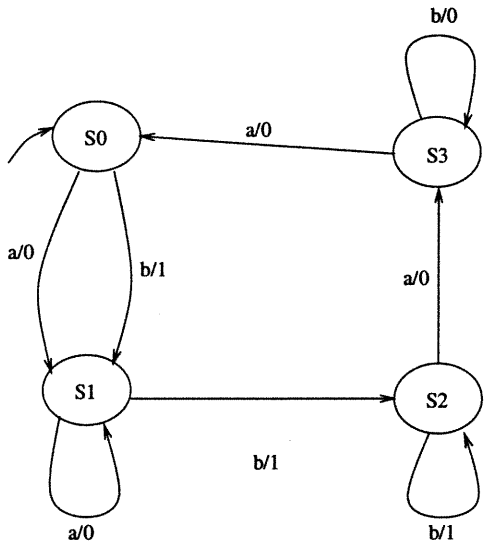

FIG. 4.6: Un exemple de MEF

État	Préambule(s_i)	Ensemble H_i	Sortie pour H_i
s_0	Nul	$\{b.a.b, a.b\}$	$\{1.0.1, 0.1\}$
s_1	$a/0$	$\{b.a.b, a.b\}$	$\{1.0.0, 0.1\}$
s_2	$a/0.b/1$	$\{a.b, b\}$	$\{0.0, 1\}$
s_3	$a/0.b/1.a/0$	$\{b\}$	$\{0\}$

TAB. 4.6: Les ensembles H_i pour la MEF de la Figure 4.6

4.2 Le test basé sur le modèle des MEFEs

Le problème fondamental de la génération de tests à partir d'une MEFE est l'*exécutabilité des chemins*. Ceci est équivalent à trouver un chemin dans lequel toutes les transitions sont exécutables. Autrement dit, le prédicat de chaque transition appartenant au chemin à générer doit être satisfait. Ce problème se pose aussi dans le test des systèmes temps réel dont la spécification contient des variables d'horloges et des contraintes sur ces variables (voir chapitre 6).

Pour générer des cas de test à partir d'une MEFE, on combine les techniques des MEFs et celles des langages de programmation (test structurel). Ceci consiste à combiner une méthode des MEFs avec les critères de flux de données et les méthodes de variation de paramètres. Cependant, la difficulté principale est que le flux de contrôle dans une MEFE est dépendant de données. Il est indécidable de déterminer une séquence d'entrées forçant l'IST à exécuter une transition donnée.

4.2.1 Analyse du flux de données

L'analyse du flux de données est basée sur la construction d'un graphe, dit *graphe du flux de données*. Ce graphe est orienté, dans lequel les noeuds représentent les unités fonctionnelles du programme et les arcs décrivent le flux de données. Une unité fonctionnelle peut être une instruction, une transition, une procédure ou même un programme. L'objectif de la construction du graphe du flux de données est de tester la dépendance de données entre les transitions. Plus spécifiquement, il s'agit de tester la dépendance entre les définitions et les utilisations subséquentes des variables. Pour ce faire, plusieurs critères [WR85] ont été définis pour être satisfaits par les cas de test générés. Ces critères portent sur les occurences des variables dans la MEFE. L'occurence d'une variable x est dite un *def* de x si x est utilisée dans la partie gauche d'une affectation ou dans une instruction de lecture. D'un autre côté, l'occurence d'une variable x est dite un *c-use* de x si x est utilisée dans la partie droite d'une affectation ou dans une instruction d'écriture. Enfin, l'occurence d'une variable

x est dite un *p-use* de x si x est utilisée dans la condition d'une transition.

Un *c-use* d'une variable x est dit *global* s'il n'existe aucun *def* de x précédant ce *c-use* dans son bloc ; sinon il est dit *local.* À chaque noeud i du graphe de flux de données, on associe deux ensembles *def(i)* et *c-use(i)* désignant respectivement l'ensemble des variables ayant un *def* dans ce noeud et l'ensemble des variables ayant un *c-use global* dans ce noeud. De même, à chaque arc (i, j), on associe un ensemble *p-use(i, j)* contenant les variables ayant un *p-use* dans cet arc. Un chemin dans un graphe du flux de données est une séquence finie de noeuds $(n_1, n_2, ..., n_k), k \geq 2$, tel qu'il existe un arc de n_i à n_{i+1} pour $i = 1, 2, ..., k-1$. Un chemin $(i, n_1, n_2, ..., n_m, j), m \geq 0$ est dit *def-clear*, de noeud i au noeud j, par rapport à une variable x s'il n'existe aucun *def* de x dans les noeuds $n_1, n_2, ..., n_m$. De même, un chemin $(i, n_1, n_2, ..., n_m, j, k), m \geq 0$ est dit *def-clear*, de noeud i à l'arc (j, k), par rapport à une variable x s'il n'existe aucun *def* de x dans les noeuds $(n_1, n_2, ..., n_m, j)$. Enfin, un chemin $(n_1, n_2, ..., n_j, n_k)$ est dit *du-path* par rapport à une variable x si n_1 contient une définition globale de x et

- soit n_k a un c-use de x et $(n_1, n_2, ..., n_j, n_k)$ est un chemin def-clear simple (tous ses noeuds sont distincts sauf éventuellement le premier et le dernier) par rapport à x, ou
- (n_j, n_k) a un p-use de x et $(n_1, n_2, ..., n_j)$ est un chemin def-clear sans boucle par rapport à x.

Ainsi, pour chaque noeud i et pour chaque variable x tel que $x \in def(i)$, *dcu(x, i)* est l'ensemble des noeuds j tel que $x \in$ *c-use(j)* et pour lesquels il existe un chemin *def-clear* de i à j ; *dpu(x, i)* est l'ensemble des arcs (j, k) tel que $x \in$ *p-use(j, k)* et pour lesquels il existe un chemin *def-clear* de i à j. Les critères du test des MEFEs sont représentés dans la Figure 4.7 et sont définis comme suit :

Soit P un ensemble de chemins d'un graphe du flux de données G.

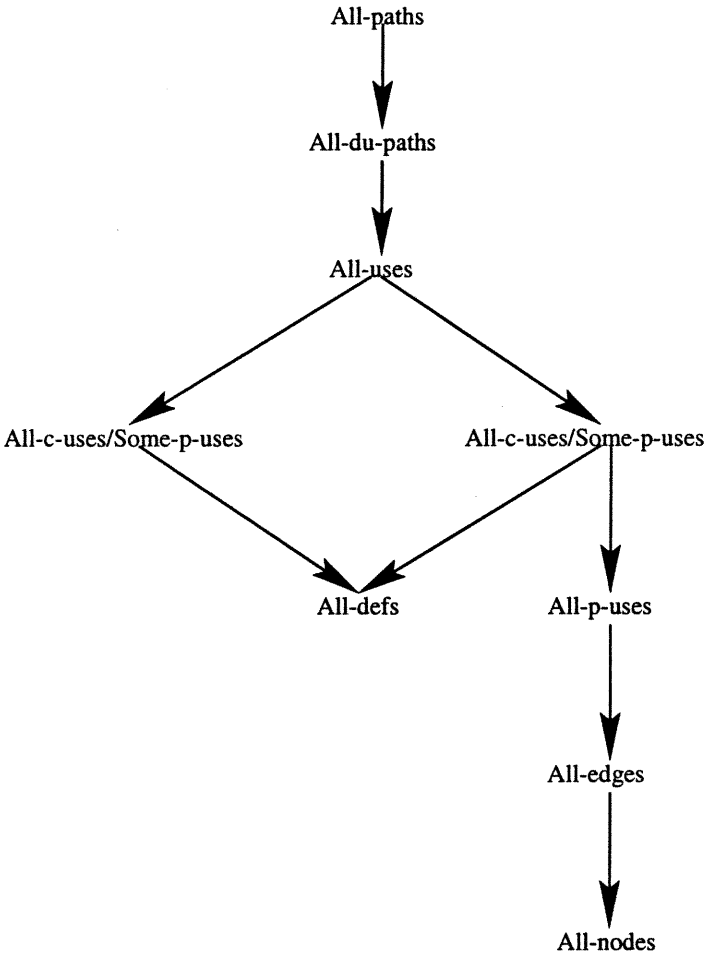

FIG. 4.7: Quelques critères du test des MEFEs

- P satisfait le critère *all-nodes* si P contient tous les noeuds de G.
- P satisfait le critère *all-edges* si P contient tous les arcs de G.
- P satisfait le critère *all-defs* si pour tout noeud i et tout $x \in def(i)$, P contient un chemin def-clear relatif à x du i à un élément de *dcu(x, i)* ou *dpu(x, i)*.
- P satisfait le critère *all-p-uses* si pour tout noeud i et tout $x \in def(i)$, P contient un chemin def-clear relatif à x du i à tous les éléments de *dpu(x, i)*.
- P satisfait le critère *all-c-uses/some-p-uses* si pour tout noeud i et tout $x \in def(i)$, P contient un chemin def-clear relatif à x du i à chaque élément de *dcu(x, i)*; si *dcu(x, i)* est vide alors P doit inclure un chemin def-clear relatif à x du i à un élément de *dpu(x, i)*.

- P satisfait le critère *all-p-uses/some-c-uses* si pour tout noeud i et tout $x \in def(i)$, P contient un chemin def-clear relatif à x du i à tous les éléments de *dpu(x, i)*; si *dpu(x, i)* est vide alors P doit inclure un chemin def-clear relatif à x du i à un élément de *dcu(x, i)*.
- P satisfait le critère *all-uses* si pour tout noeud i et tout $x \in def(i)$, P contient un chemin def-clear relatif à x du i à tous les éléments de *dcu(x, i)* et tous les éléments de *dpu(x, i)*.
- P satisfait le critère *all-du-paths* si pour tout noeud i et tout $x \in def(i)$, P contient tout du-path relatif à x.
- P satisfait le critère *all-paths* si P contient tout chemin complet (menant du noeud initial à un noeud terminal) de G.

Dans la Figure 4.7, la flèche d'un critère c_1 à un critère c_2 signifie que c_1 est plus fort que c_2 ; autrement dit, si un cas de test satisfait le critère c_1 alors il satisfera aussi le critère c_2. Le choix d'un critère ou d'un autre pour la sélection de tests nécessite toujours un compromis entre la qualité de la couverture souhaitée et la complexité et la faisabilité du test. 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. Dans la section suivante, nous présentons une brève revue des méthodes du test des MEFEs les plus connues [DSA+99, Bou99].

4.2.2 Méthodes de génération de tests à partir des MEFES

Les méthodes classiques du test des MEFs (voir section 4.1) ne sont pas suffisantes pour la génération de tests à partir d'une MEFE. La partie données doit aussi être testée pour juger le comportement de l'implantation. Durant les deux dernières décennies, certains travaux ont été réalisés pour tester le flux de données dans les protocoles de communication [UY91, HLJ95, RDT95a, BDAR97].

Dans [UY91], les auteurs présentent une méthode pour la sélection automatique

des séquences de test pour tester en même temps les flux de contrôle et de données d'un protocole de communication. La génération de tests est basée sur l'identification et la couverture de chaque association entre une sortie et toutes les entrées qui influencent cette sortie. La méthode exige que chaque association soit examinée au moins une fois durant le test.

Dans [HLJ95], les auteurs proposent une méthode pour générer des cas de test exécutables pour les flux de données et de contrôle d'un protocole de communication spécifié sous forme d'une MEFE. La méthode se base sur une sorte d'analyse d'accessibilité et utilise le critère *toutes les définition-sorties* pour tester le flux de données.

Dans [RDT95a], Ramalingom et al. présentent une méthode unifiée de génération de cas de test pour les protocoles de communication modélisés par des MEFEs en utilisant les séquences uniques indépendantes du contexte (CUIS). La méthode résout le problème d'exécutabilité des cas de test pendant leur génération. Un nouveau type de séquences d'identification, $Trans - CUIS$, est défini. Le critère $Trans - CUIS$ consiste à générer une séquence unique d'entrées 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.

Dans [BDAR97], les auteurs présentent une méthode pour la génération des cas de test exécutables à partir d'une spécification donnée sous forme d'une MEFE. Les cas de test obtenus couvrent en même temps le flux de contrôle et le flux de données du protocole en question. La méthode utilise la séquence *UIO* (voir section 4.1) pour tester le flux de contrôle et le critère *tous-définition-usages* pour vérifier le flux de données. L'exécutabilité des chemins definition-usages est assurée par une interprétation symbolique de toutes les variables de chaque prédicat en montant jusqu'à ce que ces variables soient représentées par des constantes et des paramètres d'entrée. Si le prédicat est faux, la méthode applique une heuristique pour rendre le chemin

exécutable.

4.3 Le test des systèmes temps réel

Les trois dernières décennies ont connu une intense activité de recherche dans le domaine du test non temporisé. Cependant, les systèmes informatiques sont actuellement de plus en plus spécifiés avec des contraintes temporelles conditionnant leurs exécutions. Ceci a poussé les chercheurs à développer de nouvelles méthodes de test pour ces systèmes. Dans ce qui suit, nous passons en revue la liste des travaux qui ont été réalisés pour générer des cas de test temporisés.

4.3.1 Génération de tests à partir des formules logiques

Cette approche [MMM95] consiste à générer des cas de test à partir d'une formule logique. La logique utilisée est une extension de la logique temporelle classique pour modéliser les systèmes temps réel. L'aspect temporel est exprimé à l'aide de deux constructeurs de base, *Futur* et *Past*, referant respectivement à des instants dans le futur et dans le passé. Dans cette logique, une seule horloge est utilisée.

Pour générer des cas de test pour une formule F, F est décomposée d'une manière hiérarchique en des formules atomiques ne contenant que des *événements*. Un événement est un couple (L, t) formé du litéral L et de l'instant t. Un ensemble d'événements qui ne contient pas de contradiction (en même temps les deux événements (L, t) et $(\neg L, t)$) est appelé *histoire*. Chaque feuille ne contenant aucune contradiction dans l'arbre de décomposition d'une formule F est considérée comme une histoire de F. Une histoire d'une formule F est dite *complète* si elle contient une valeur de vérité unique pour chaque prédicat de la formule à tout point du domaine temporel. Un cas de test d'une formule F est une histoire complète satisfaisant F à un ou plusieurs points de son domaine temporel. Ce domaine temporel est discrétisé en des valeurs entières.

Un cas de test complet d'une formule F est construit par l'application des opérations de *décalage* et de *concaténation* sur des cas de test de tailles réduites, dits *cas de test élémentaires.* Ceux-ci sont extraits directement de l'arbre de décomposition de F. Le décalage de Δ unités de temps d'un cas de test satisfaisant une formule F à l'instant t consiste en un cas de test satisfaisant F à l'instant $t+\Delta$. Par contre, la concaténation de deux cas de test TC_1 et TC_2 est le cas de test TC_3 obtenu par l'union des ensembles d'événements de TC_1 et TC_2.

Cette façon de générer des cas de test temporisés souffre des lacunes suivantes. La logique utilisée n'est pas, à notre sens, suffisamment adéquate pour la génération de tests. De plus, la suite de test générée ne peut couvrir que les valeurs entières de domaines temporels des formules logiques de la spécification. Enfin, la logique utilisée ne peut servir que pour décrire une classe très restreinte des systèmes temps réel (ceux qui sont spécifiés à l'aide d'une seule horloge).

4.3.2 Génération de tests à partir d'une MEF avec des temporisateurs et des compteurs

Cette méthode [Liu93] génère des cas de test temporisés à partir d'une MEF avec des temporisateurs et des compteurs. Elle utilise la méthode Wp [FBK+91] en procédant comme suit. Premièrement, le comportement de chaque temporisateur et celui de chaque compteur sont modélisés par des MEFs. Ensuite, tous les MEFs sont composées pour n'avoir qu'une seule MEF globale décrivant tout le système. Enfin, la méthode Wp est appliquée sur la MEF globale avec certaines hypothèses.

La MEF d'un temporisateur est définie par :

- Deux états indiquant respectivement l'*activité* et l'*inactivité* du temporisateur.
- Un alphabet formé des événements correspondant respectivement aux déclenchement, l'arrêt et l'expiration du temporisateur, et le passage de la durée pour

laquelle est activé le temporisateur.

- Un ensemble de transitions entre les états sur les événements appropriés.

La MEF d'un compteur, quant à elle, est définie par :

- Un ensemble d'états représentant les différentes valeurs du compteur.
- Deux événements représentants respectivement la remise à zéro du compteur et son incrémentation.
- Un ensemble de transitions entre les états sur les événements appropriés.

Le problème de cette approche est qu'elle n'est pas applicable aux systèmes avec une forme générale de contraintes temporelles.

4.3.3 Génération de tests à partir d'un graphe de contraintes

La spécification utilisée dans cette approche [CL97, Cla96] est donnée sous forme d'un graphe de contraintes. À partir du modèle de fautes, les auteurs définissent certains *critères de test* et génèrent des cas de test qui les satisfont. Ces critères, qui sont illustrés dans la Figure 4.8, ressemblent en quelques sorte aux critères utilisés pour tester les MEFEs. Une flèche d'un critère C_1 vers un critère C_2 signifie que le critère C_1 inclut le critère C_2 (lorsque le critère C_1 est satisfait le critère C_2 l'est aussi). L'inclusion marquée par une astérisque n'est vraie que dans des cas particuliers.

Le critère de test *All-Input-Delays* nécessite que chaque point du domaine temporel d'une contrainte soit testé. Cependant, ce critère est impossible à satisfaire du fait que le domaine temporel des contraintes est dense. Par conséquent, une approximation de ce critère s'impose et donne lieu au critère *All-Input-Bounds*. Ce dernier consiste à tester les bornes de chaque contrainte de la spécification. Ce critère est, à son tour, irréalisable pour les contraintes données sous forme d'intervalles ouverts. De ce fait, le critère *All-Input-Bounds-Approximate* est défini et consiste à tester la

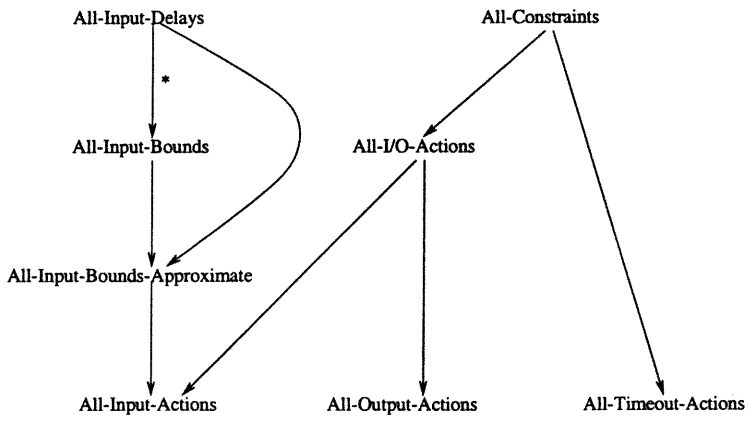

FIG. 4.8: Les critères de test d'un graphe de contraintes

contrainte à l'intérieur de l'intervalle, au voisinage de la borne supérieure et la borne inférieure. Le critère *All-Input-Actions* exige qu'au moins un point du domaine temporel de chaque entrée f soit couvert par le test.

En outre, afin d'avoir un test complet du domaine temporel d'une entrée, le critère *All-Timeout-Actions* est défini. Ce critère est dit satisfait par un ensemble de valeurs temporelles réalisables D si et seulement si pour chaque entrée f, il existe au moins une valeur dans D qui amène le système à un état où f est applicable et retient f durant tout l'intervalle $dom(f)$. Puisque les sorties d'un système ne sont pas contrôlables, il existe un seul critère de test pour les sorties, dit *All-Output-Actions.* Un ensemble de valeurs temporelles réalisables D satisfait le critère *All-Output-Actions* si et seulement si pour chaque sortie f, il existe au moins une valeur dans D qui amène le système à un état où f doit être observée et vérifie que f est produite à l'intérieur de l'intervalle $dom(f)$. Pour unifier tous ces critères, les critères *All-I/O-Actions* et *All-Constraints* sont définis. le critère *All-I/O-Actions* est satisfait si et seulement si les deux critères *All-Input-Actions* et *All-Output-Actions* sont satisfaits. Le critère *All-Constraints*, quant à lui, est satisfait si et seulement si les deux critères *All-I/O-Actions* et *All-Timeout-Actions* sont satisfaits.

L'approche proposée a deux limitations. La première concerne le modèle (c'est-à-

Le résultat principal de ce travail est la preuve de la possibilité d'un test exhaustif pour un automate temporisé par rapport à une relation de bisimulation. En fait, toute implantation, passant avec succès la suite de test générée, est déclarée *bisimilaire* à la spécification. Cependant, les problèmes majeurs de cette méthode sont la restriction du modèle utilisé et le nombre de cas de test générés. Le modèle utilisé est un AT particulier dans lequel les sorties du système ne peuvent avoir lieu qu'à des valeurs entières de l'espace temporel des horloges. Cette restriction ne permet pas de décrire une large gamme des systèmes temps réel dont les sorties peuvent être produites à l'intérieur d'une plage temporelle et non pas à un point bien déterminé. Quant à la suite de test générée, le nombre de cas de test la constituant est très grand même pour un exemple académique simple. En fait, ce grand nombre de cas de test est dû à la durée des transitions de délai 2^{-n} utilisée dans la construction de l'automate de grille. Cette durée est très petite pour deux raisons. D'une part, le nombre de régions d'horloges dans un AT est exponentiel en nombre d'horloges et les constantes utilisées dans les contraintes sur ces horloges. D'autre part, les auteurs n'utilisent pas le nombre exact de régions d'horloges mais plutôt une approximation par une borne supérieure donnée par Alur et Dill [AD94].

4.4 Conclusion

Dans ce chapitre, nous avons passé en revue les méthodes de génération de cas de test à partir des modèles formels les plus utilisés. Plus précisément, nous avons présenté le test basé sur les MEFs, le test fondé sur les MEFEs et le test temporisé. Pour chacune des méthodes présentées, nous avons expliqué ses avantages, ses inconvénients et les conditions de son application. Certaines de ces méthodes seront utilisées dans les prochains chapitres pour être appliquées dans nos approches (par exemple la méthode Wp généralisée) ou comparées avec nos méthodes de génération de tests (les méthodes du test temporisé).

Dans le reste de cette thèse, nous abordons plus profondément le test des systèmes temps réel modélisés par des automates à entrées sorties temporisées et nous présentons nos contributions majeures. Pour ce faire, le prochain chapitre sera d'abord consacré à la présentation du modèle des automates à entrées sorties temporisées et à la discussion des problèmes rencontrés lors du test des systèmes temps réel en faisant référence au test des MEFs et des MEFEs.

Chapitre 5

Choix du modèle et discussion des problèmes

Lors du chapitre précédent, nous avons passé en revue certaines méthodes de génération de cas de test pour les systèmes temps réel modélisés par différents formalismes. Ces méthodes souffrent de certains problèmes tels que la restriction des modèles utilisés et/ou la complexité et la non-applicabilité des approches présentées. Ceci nous pousse à développer des méthodes efficaces et pratiques pour tester les systèmes temps réel, tout en se basant sur un modèle adéquat et général de spécification. Dans ce chapitre, nous présentons d'abord le modèle des Automates à Entrées Sorties Temporisées (AESTs) que nous utilisons comme base pour la génération des cas de test temporisés. Ensuite, nous discutons les problèmes du test des systèmes temps réel modélisés par des AESTs tout en le comparant au test des MEFs et des MEFEs.

5.1 Les automates à entrées sorties temporisées

Les automates à entrées sorties temporisées [ENDKE98, SVD01] sont une variante des automates temporisés d'Alur et Dill où l'ensemble d'alphabet est subdivisé en un ensemble d'entrées représentant les messages que le système peut accepter de son environnement, et un ensemble de sorties désignant les messages que le système

envoie à son environnement. Formellement, un automate à entrées sorties temporisées est défini comme suit.

Définition 5.1 *Automates à entrées sorties temporisées*
Un automate à entrées sorties temporisées (AEST) est un tuple (I, O, L, l_0, C, T), *où :*

- I *est un ensemble fini d'entrées que l'automate reçoit de son environnement. Chaque entrée commence par "?".*
- O *est un ensemble fini de sorties que l'automate envoie à son environnement. Chaque sortie commence par "!".*
- L *est un ensemble fini d'emplacements.*
- $l_0 \in L$ *est l'emplacement initial.*
- C *est un ensemble fini d'horloges initialisées toutes à zéro en* l_0. *Le domaine temporel de chaque horloge* x *est* $[0, C_x] \cup \{\infty\}$, *où* C_x *est la plus grande constante entière utilisée dans les contraintes sur l'horloge* x.
- T *est un ensemble fini de transitions. Ces dernières sont formées de la même manière que les transitions dans un automate temporisé (chapitre 3).*

Dans cette définition, la valeur de toute horloge $x \in C$ n'est pertinente qu'au dessous de la constante C_x. C'est pourquoi, nous représentons toutes les valeurs supérieures à C_x par la constante ∞ et nous écrivons : $\forall \epsilon > 0, \forall x \in C, C_x + \epsilon = \infty$.

La Figure 5.1 montre un exemple d'AEST avec deux horloges x et y. L'automate a deux emplacements l_0 (l'emplacement initial) et l_1 et trois transitions entre ces emplacements. Par exemple, la transition $l_1 \xrightarrow{?In,\{x,y\},x<1\&y<1}_A l_0$ est tirable sur l'entrée In lorsque les horloges x et y sont inférieures strictement à 1, remet à zéro les horloges x et y et ramène la machine à son emplacement initial. Le domaine temporel des deux horloges x et y est le même et il est égal à $[0, 1] \cup \{\infty\}$.

Tout comme pour les automates temporisés, la sémantique d'un AEST est donnée par un système de transitions étiquetées temporisé. Ce dernier est construit à partir

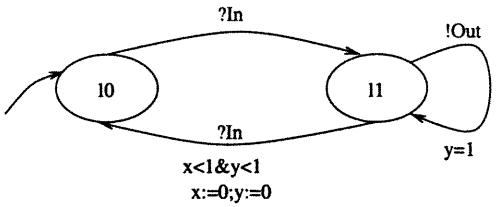

FIG. 5.1: Un exemple d'AEST.

des notions d'état et d'interprétations d'horloges et contient les transitions explicites et les transitions de délai (voir section 3.2.5). Cependant, ce système de transitions étiquetées est infini à cause de l'infinité de transitions de délai ou de l'espace d'états. Pour réduire cet espace, on définit une relation d'équivalence sur l'ensemble d'interprétations d'horloges $V(C)$ (voir section 3.2.5) de la manière suivante [AD94].

Définition 5.2 *Régions d'horloges*
Soit un AEST $A = (I, O, L, l_0, C, T)$ *tel que le domaine temporel de chaque horloge* $x \in C$ *est* $[0, C_x] \cup \{\infty\}$. *Deux interprétations d'horloges* v *et* v' *sont dites équivalentes, que l'on note* $v \sim v'$, *si et seulement si :*

- $\forall x \in C, \lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$.
- $\forall x, y \in C \mid ((v(x) \neq \infty) \wedge (v(y) \neq \infty)), (fract(v(x)) \leq fract(v(y)) \Leftrightarrow fract(v'(x)) \leq fract(v'(y)))$.
- $\forall x \in C \mid v(x) \neq \infty, (fract(v(x)) = 0 \Leftrightarrow fract(v'(x)) = 0)$.
  où $\lfloor x \rfloor$ *et* $fract(x)$ *représentent respectivement la partie entière et la partie fractionnaire du nombre* x.

Une région d'horloges d'un *AEST* A est une classe d'équivalence d'interprétations d'horloges induite par $\sim$. $[v]$ désigne la région d'horloges dont v fait partie. Si nous reprenons l'exemple de la Figure 5.1, nous obtenons l'ensemble des régions d'horloges de la Figure 5.2. Sur cette figure, il y a dix-huit régions différentes. Chacune de ses régions est identifiée par un numéro ($R1, R2, ..., R18$) et caractérisée par un ensemble d'inéquations sur les horloges utilisées (par exemple, $R16 : 0 < x < y < 1$). D'un

autre côté, nous distinguons trois types de régions sur la Figure 5.1 : les points corners $\{R1, R2, R3, R4\}$, les segments ouverts $\{R5, R6, R7, R8, R9, R10, R11, R12, R13\}$ et les régions ouvertes $\{R14, R15, R16, R17, R18\}$.

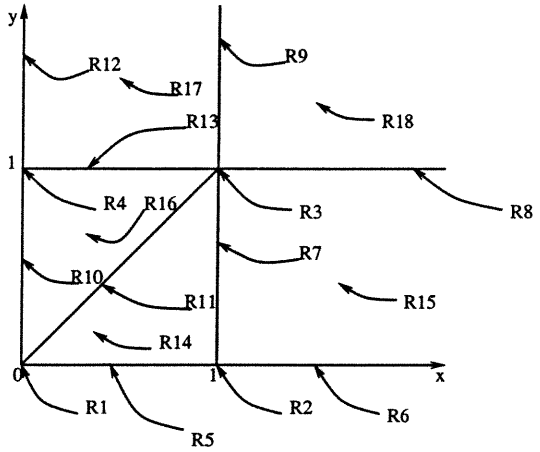

FIG. 5.2: Un exemple de régions d'horloges.

Une borne supérieure du nombre de régions d'horloges dans un *AEST* est donnée dans [AD94]. Ce nombre est de très loin plus grand que le nombre exact calculé dans [EEN98].

En se basant sur la définition 5.2, nous représentons le modèle sémantique d'un *AEST* A par un automate fini, dit *automate des régions* ou *graphe des régions.*

Définition 5.3 *Graphe des régions*
Soit $A = (I, O, L, l_0, C, T)$ *un AEST. L'automate de régions de* A *est un automate* $RA = (\Sigma, S, s_0, T)$, *où :*

- $\Sigma = I \cup O \cup \mathbf{R}^{>0}$.
- $S = \{\langle l, [v]\rangle \mid l \in L \wedge v \in V(C)\}$.
- $s_0 = \langle l_0, [v_0]\rangle$, *où* $v_0(x) = 0$ *pour toute* $x \in C$.

- *RA a une transition* , $s \xrightarrow{a} s'$, *de l'état* $s = \langle l, [v] \rangle$ *à l'état* $s' = \langle l', [v'] \rangle$ *sur l'action* $a \in I \cup O$, *si et seulement s'il existe une transition* $l \xrightarrow{a,\lambda,G} l'$ *tel que* $v \models G$ *et* $v' = [\lambda := 0]v$.
- *RA a une transition de délai,* $s \xrightarrow{d} s'$, *de l'état* $s = \langle l, [v] \rangle$ *à l'état* $s' = \langle l, [v'] \rangle$ *sur un délai* $d \in \mathbf{R}^{>0}$, *si et seulement si* $[v'] = [v + d]$.

L'automate des régions est au coeur de la vérification et du test des systèmes temps réel car il représente toutes les traces possibles du système décrit par l'*AEST*. La définition de l'automate des régions que nous avons donnée ici est légèrement différente de celle présentée dans [AD94] du fait qu'elle explicite les transitions de délai.

La Figure 5.3 montre le graphe de régions de l'AEST de la Figure 5.1. Chaque état de ce graphe est formé d'un emplacement de l'automate d'origine de la Figure 5.1 et d'une région d'horloges de la Figure 5.2. Certaines régions d'horloges sont inaccessibles et ne figurent pas donc sur le graphe. Quant aux transitions du graphe, elles sont formées des transitions explicites sur des entrées/sorties et des transitions de délai (étiquetées par le symbole d sur la figure).

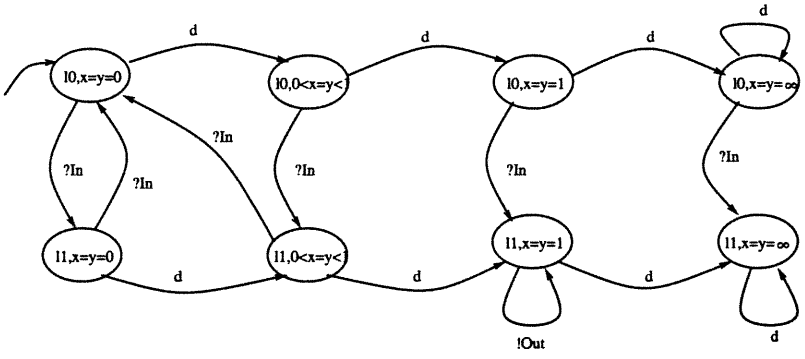

FIG. 5.3: Le graphe de régions de l'AEST de la Figure 5.1.

5.2 Problématique du test des systèmes temps réel

Le problème fondamental du test d'un système temps réel est l'existence des variables d'horloges dans la spécification. Ceci dit qu'en plus de l'observation des sorties et la vérification des états d'arrivée du système, il faut aussi vérifier si l'implantation du système n'accepte des entrées (respectivement ne répond par des sorties) que dans l'intervalle de temps fixé dans la spécification. Ceci est difficile car le temps n'est pas sous un contrôle direct de l'utilisateur (testeur). De plus, cette difficulté varie selon la sémantique du temps adoptée et le modèle de spécification utilisé. Par exemple, les systèmes à temps discret sont beaucoup plus faciles à tester que les systèmes à temps continu ou dense. De même, le test d'un modèle général tel qu'un AEST est plus délicat que celui d'un modèle simple tel qu'une logique temporelle avec une seule variable d'horloges. Dans ce qui suit, nous nous intéresserons à la génération des cas de test à partir d'un AEST.

De point de vue test, le modèle des AESTs peut être approché à une MEF ou une MEFE. Si on considère le temps comme une entrée du système, on pourra appliquer les techniques des MEFs. En revanche, si on voit les variables d'horloges comme des variables de données dans une MEFE, on pourra alors adapter les techniques du test des MEFEs pour les appliquer aux AESTs. Bref, pour tester un AEST, les points suivants doivent, entre autres, être traîtés :

- Le problème d'*exécutabilité des chemins* dû aux variables d'horloges.
- Le modèle de fautes.
- La couverture de fautes.
- La relation de conformité.
- La densité du domaine temporel.

L'exécutabilité des chemins est un problème qui se pose lors de la génération des cas de test à partir de tout modèle enrichi par des variables et des prédicats sur ces

variables. Ce problème consiste à générer des cas de test assurant que le prédicat de chaque transition traversée par chacun d'eux soit satisfait. Pour résoudre le problème d'exécutabilité des chemins dans un AEST, nous utilisons le graphe de régions comme sémantique d'AEST. Si s_j est l'état d'arrivée d'une transition sortante de l'état s_i dans le graphe de régions alors la variation des valeurs de différentes horloges, entre s_i et s_j, est la même. Ainsi, le problème d'exécutabilité des chemins est résolu en faisant varier minutieusement les valeurs d'horloges entre chaque paire d'états consécutifs (s_i, s_j).

Le modèle de fautes, quant à lui, est très important pour développer une méthode efficace de génération de test. Dans le cas des systèmes temps réel, le modèle de fautes inclut le modèle de fautes des FSMs et les fautes dues au non respect des contraintes temporelles.

La couverture de fautes est l'un des critères utilisés pour juger l'efficacité et/ou comparer des méthodes de génération de test pour un système donné. Elle permet de savoir quelles sont les fautes détectables et celles qui restent indétectables par les cas de test soumis à une implantation fautive. Dans le cas des systèmes temps réel, le développement d'une méthode avec une bonne couverture de fautes est difficile surtout lorsqu'on veut détecter toutes les fautes dues à la violation des contraintes temporelles.

Quant à la relation de conformité, il est clair qu'une équivalence de traces entre la spécification et l'implantation est la meilleure à utiliser. Cependant, l'équivalence de traces entre deux automates temporisés est *indécidable*[ACH94]. Ceci est principalement dû à l'infinité de transitions de délai dans un automate temporisé. Autrement dit, l'indécidabilité de l'équivalence de traces est à cause de la densité du domaine temporel des horloges dans l'AEST. Pour remédier à ce problème, on doit utiliser une relation moins forte qu'une équivalence de traces pour tester une implantation d'un système temps réel par rapport à sa spécification. Nous verrons dans les chapitres à

venir comment se contenter d'un sous-ensemble fini du domaine temporel et tester l'implantation en ces points. Ce sous-ensemble est choisi de telle sorte à ce que chaque région d'horloges, dans le graphe de régions, soit représentée par au moins un élément dans le sous-ensemble.

5.3 Conclusion

Dans ce chapitre, nous avons commencé par présenter le modèle des AESTs aussi bien au niveau syntaxique qu'au niveau sémantique. Ce modèle est utilisé pour spécifier les systèmes temps réel pour lesquels nous générons les cas de test. Ensuite, nous avons énuméré et discuté les différents points et problèmes à aborder lors de la génération des cas de test à partir d'un AEST. Ce sont ces points que nous allons étudier dans les chapitres à venir. Dans le prochain chapitre, nous présentons notre modèle de fautes temporelles. Ce dernier nous a servi d'une part au développement de nos méthodes de génération de cas test temporisés et d'autre part à l'étude de la couverture de fautes de ces méthodes.

Chapitre 6

Modèle de fautes temporelles

Les systèmes temps réel interagissent avec leurs environnements en utilisant des signaux d'entrée sortie sous des contraintes temporelles. Celles-ci spécifient les plages temporelles où le système en question peut accepter une entrée ou produire une sortie. Comme nous l'avons déjà mentionné, le mauvais fonctionnement de ces systèmes est généralement dû au non respect de ces contraintes temporelles et peut engendrer des conséquences catastrophiques [MMM95]. Le test est une des techniques formelles permettant de s'assurer du bon fonctionnement des systèmes temps réel. Parmi les aspects importants du test, il y a le modèle de fautes qui permet d'étudier toutes les fautes susceptibles d'exister dans une implantation d'un système. Le modèle de fautes est très dépendant du modèle formel de l'implantation.

Ce chapitre est consacré au modèle de fautes des systèmes temps réel modélisés par des automates à entrées sorties temporisées (AESTs). Nous illustrons notre modèle de fautes en prenant comme exemple de spécification l'automate de la Figure 6.1.

[1]Les résultats de ce chapitre ont été publiés dans [ENKD99a, ENKD99b].

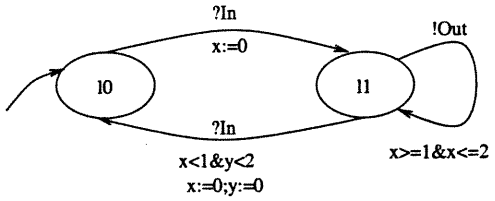

FIG. 6.1: Un exemple d'AEST avec deux horloges.

6.1 Types de fautes

Le modèle de fautes pour les systèmes temps réel modélisés par des AESTs est formé de deux catégories de fautes :

- Les fautes temporelles, et
- Les fautes de transfert et d'actions.

En outre, nous distinguons entre deux types de fautes temporelles : les fautes temporelles *effectives* et les fautes temporelles *non effectives*. Une faute temporelle est dite effective si elle a un effet sur l'exécution du système ; sinon, elle est dite non effective. Autrement dit, une faute temporelle est considérée effective si et seulement si elle change le graphe de régions du système, à l'exception de certaines transitions sur les sorties (voir section 6.2.2).

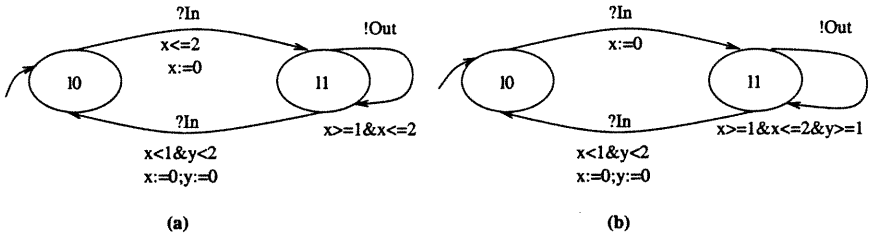

FIG. 6.2: Fautes temporelles : (a) Effectives, (b) Non effectives.

La Figure 6.2 illustre les deux types de fautes temporelles pour l'automate à entrées sorties temporisées de la Figure 6.1. L'automate de gauche (Figure 6.2 (a))

restreint la contrainte temporelle de la transition de l_0 à l_1 dans le sens où l'implantation ne peut pas accepter l'entrée $?In$ quand la valeur de l'horloge x est supérieure à 2. Cette restriction affecte l'exécution de l'implantation en modifiant son graphe de régions. Ce dernier peut avoir moins d'états et/ou de transitions que le graphe de régions de la spécification. Quant à l'automate de droite (Figure 6.2 (b)), il montre une modification de la contrainte temporelle de la transition boucle sur l_1. Ce changement de contrainte n'a aucun effet sur l'exécution de l'implantation puisque son graphe de régions reste inchangé. En effet, si la contrainte $x \geq 1$ est satisfaite dans l'emplacement l_1, la contrainte $y \geq 1$ le sera aussi car la valeur de l'horloge y est toujours supérieure ou égale à celle de l'horloge x dans l_1. Ainsi, la contrainte temporelle $x \geq 1 \& x \leq 2 \& y \geq 1$ est équivalente à la contrainte $x \geq 1 \& x \leq 2$.

Dans le reste de cette thèse, nous nous intéresserons uniquement aux fautes temporelles effectives et nous les appellerons simplement fautes temporelles.

6.2 Fautes temporelles

Les fautes temporelles sont dues à la violation des contraintes temporelles des transitions d'un AEST. Une faute temporelle peut être liée soit à la remise à zéro d'une horloge, soit à la restriction d'une contrainte d'une transition, ou soit à l'élargissement d'une contrainte d'une transition.

6.2.1 Fautes de remise à zéro des horloges

Ce type de fautes peut avoir lieu lorsqu'une implantation remet à zéro une horloge qui doit rester inchangée ou lorsqu'elle ne réinitialise pas une horloge qui doit être remise à zéro suite à une transition. Pour illustrer ce type de fautes, considérons de nouveau l'exemple de la Figure 6.1. Les Figures 6.3 et 6.4 montrent deux implantations fautives contenant les deux types de fautes de remise à zéro des horloges. L'implantation de la Figure 6.3 ne remet pas à zéro l'horloge x dans la transition

de l_0 à l_1. Cependant, l'implantation de la Figure 6.4 réinitialise l'horloge y dans la transition de l_1 à l_1.

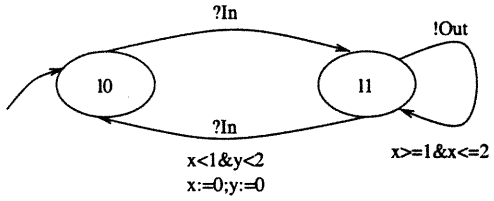

FIG. 6.3: Non remise à zéro d'une horloge.

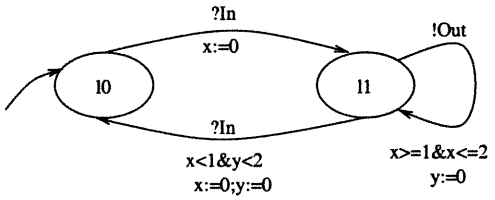

FIG. 6.4: Remise à zéro d'une horloge.

Les fautes de remise à zéro des horloges ont comme effet le changement de l'ordre entre les horloges dans le graphe de régions de l'implantation. Dans l'exemple que nous venons de voir dans le paragraphe précédent, l'implantation de la Figure 6.3 change l'ordre entre les horloges dans l'emplacement l_1. Au lieu d'avoir $x \leq y$, comme dans la spécification, on a $x = y$ car l'horloge x, tout comme l'horloge y, n'a pas été réinitialisée depuis son initialisation à l'emplacement initial. Quant à l'implantation de la Figure 6.4, elle change l'ordre entre les horloges dans l'emplacement l_1 de $x < y$ à $x \geq y$.

En ce qui concerne le nombre d'états dans le graphe des régions, nous pouvons faire les deux remarques suivantes :

- Quand une implantation n'implante pas la remise à zéro d'une horloge dans une transition, le nombre d'états dans son graphe de régions se voit diminuer. Ceci signifie que certains états du système deviennent inaccessibles. Dans l'exemple de la Figure 6.3, l'état $(l_0, 1 < x < y < 2)$ est inaccessible et le graphe de régions de l'implantation contient 12 états au lieu de 32.
- Quand une implantation remet à zéro une horloge qui doit rester inchangée, le nombre d'états dans son graphe de régions se voit augmenter. Cela veut dire que certains états du système deviennent accessibles. À titre d'exemple, le nombre d'états dans le graphe de régions de l'implantation de la Figure 6.4 est 44 au lieu de 32 et l'état $(l_1, x = 1 \& y = 0)$ est accessible.

6.2.2 Fautes de restriction des contraintes

Nous distinguons entre la restriction des contraintes pour les entrées et la restriction des contraintes pour les sorties. Puisque les entrées du système sont sous le contrôle de l'environnement, toute implantation qui rejette une entrée dont la contrainte, selon la spécification, est satisfaite est considérée fautive. Cependant, toute implantation qui restreint la contrainte temporelle d'une sortie est vue comme une réduction valide de la spécification. Bien évidemment, ceci est acceptable puisque les sorties sont contrôlées par le système et non pas par l'environnement. Pour clarifier davantage ces propos, reconsidérons la spécification de la Figure 6.1. Les Figures 6.5 et 6.6 montrent respectivement la restriction des contraintes pour les entrées et les sorties.

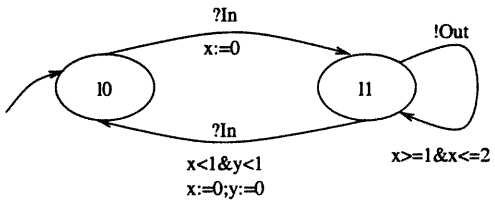

FIG. 6.5: Restriction de la contrainte d'une entrée.

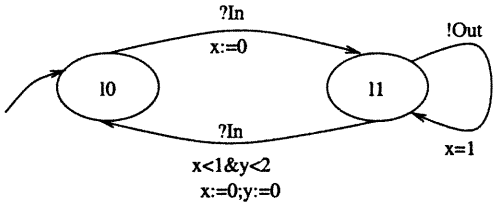

FIG. 6.6: Restriction de la contrainte d'une sortie.

L'implantation de la Figure 6.5 restreint la contrainte temporelle de la transition de l_1 à l_0 sur l'entrée $?In$. Contrairement à la spécification, cette implantation n'accepte pas l'entrée $?In$ quand la valeur de l'horloge x est inférieure à 1 et la valeur de l'horloge y est entre 1 et 2. Comme nous l'avons précédemment mentionné, cette restriction réduit le nombre d'états et de transitions dans le graphe de régions de l'implantation, comparativement à celui de la spécification. Dans notre exemple de la Figure 6.5, le nombre d'états dans le graphe de régions de l'implantation est 23 alors que le nombre d'états dans le graphe de régions de la spécification est 32.

D'un autre côté, l'implantation de la Figure 6.6 restreint la contrainte temporelle de la transition de l_1 à l_1 sur la sortie $!Out$. Cette implantation répond toujours par la sortie $!Out$ avant que la valeur de l'horloge x n'atteint 2. Comme les sorties sont contrôlées par l'implantation et que l'essentiel pour le testeur est de les observer dans les plages temporelles permises, la contrainte de la transition dans la spécification est alors satisfaite. De ce fait, l'implantation n'est plus considérée fautive mais elle est plutôt vue comme une réduction valide de la spécification.

6.2.3 Fautes d'élargissement des contraintes

Les fautes d'élargissement des contraintes sont liées à la modification des bornes des contraintes temporelles dans les transitions. Lorsqu'une implantation élargit la borne supérieure d'une contrainte ou lorsqu'elle réduit sa borne inférieure, elle est considérée fautive. Plus précisément, une faute d'élargissement d'une contrainte peut

se produire dans l'un des cas suivants :

- L'implantation remplace une contrainte $x < m_1$ par la contrainte $x < m'_1$ tel que $m'_1 > m_1$ et $m'_1 \in [0, C_x]$.
- L'implantation remplace une contrainte $x = m_2$ par la contrainte $x > m'_2$ tel que $m'_2 < m_2$.
- L'implantation remplace une contrainte $x = m_3$ par la contrainte $x < m'_3$ tel que $m'_3 > m_3$ et $m'_3 \in [0, C_x]$.
- L'implantation remplace une contrainte $x > m_4$ par la contrainte $x > m'_4$ tel que $m'_4 < m_4$.

où x est une horloge et $m_1, m_2, m_3, m_4, m'_1, m'_2, m'_3$ et m'_4 sont des entiers naturels.

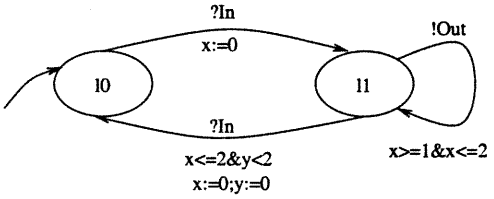

FIG. 6.7: Élargissement de la contrainte d'une entrée.

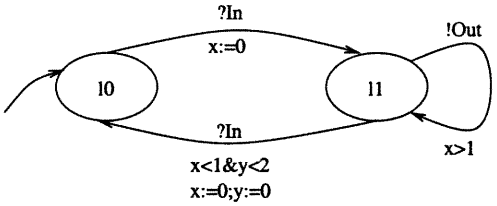

FIG. 6.8: Élargissement de la contrainte d'une sortie.

Contrairement à la restriction des contraintes, l'élargissement d'une contrainte est considérée comme une faute dans le cas des entrées et des sorties. Les Figures 6.7 et 6.8 montrent deux implantations fautives pour la spécification de la Figure 6.1.

L'implantation de la Figure 6.7 élargit la contrainte de la transition de l_1 à l_0 sur l'entrée $?In$. Contrairement à la spécification, cette implantation accepte l'entrée $?In$ même si la valeur de l'horloge x est entre 1 et 2 et celle de l'horloge y est inférieure à 2. L'implantation de la Figure 6.8, de son côté, élargit la contrainte de la transition de l_1 à l_1 sur la sortie $!Out$. Cette implantation peut répondre par la sortie $!Out$ même lorsque la valeur de l'horloge x est supérieure strictement à 2.

En outre, l'élargissement des contraintes peut augmenter le nombre d'états et /ou de transitions dans le graphe de régions de l'implantation du système, comparativement au graphe de régions de sa spécification. Par exemple, dans le cas de la Figure 6.7, le nombre d'états dans le graphe de régions reste inchangé (le même que celui de la spécification) mais certaines transitions sont rajoutées tel que la transition $(l_1, 1 < x = y < 2) \xrightarrow{?In} (l_0, x = y = 0)$.

6.3 Fautes de transfert et d'actions

Ces fautes sont similaires aux fautes de sorties et de transfert dans les MEFs et les STEs [BDD+92]. Une implantation est dite avoir une faute de sortie si elle ne répond pas par la sortie attendue dans l'un de ses états. Par contre, une implantation est dite avoir une faute de transfert si, à la suite de l'exécution d'une transition sur une entrée ou une sortie, elle passe à un état différent de celui attendu.

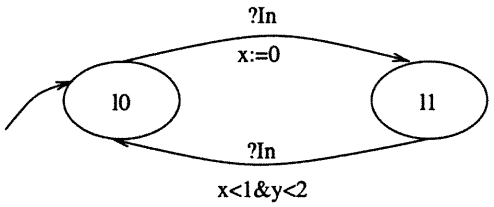

FIG. 6.9: Un exemple de fautes de sorties.

Les Figures 6.9 et 6.10 illustrent respectivement ces deux types de fautes. L'im-

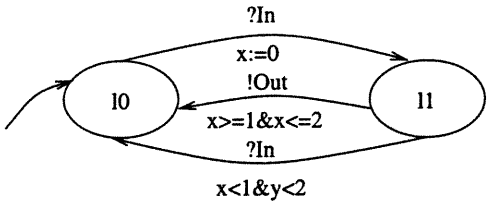

FIG. 6.10: Un exemple de fautes de transfert.

plantation de la Figure 6.9 contient une faute de sortie dans l_1. Elle ne répond pas par la sortie $!Out$ après l'application de l'ensemble de séquences $\{\frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}, \frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}, \frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}, \frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}, \frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}\}$. Par ailleurs, l'implantation de la Figure 6.10 contient une faute de transfert dans l'état $(l_1, x = 1 \& y = 5/4)$. En effet, après l'exécution de la séquence $\frac{1}{4}.?In.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.!Out$, l'implantation entre dans l'état $(l_0, x = 1 \& y = \frac{5}{4})$ au lieu de l'état $(l_1, x = 1 \& y = \frac{5}{4})$.

6.4 Conclusion

Dans ce chapitre, nous avons vu le modèle de fautes temporelles basé sur les AESTs. Ce modèle regroupe toutes les fautes simples qu'on peut trouver dans une implantation modélisée par un AEST. Il joue un rôle important dans la génération des cas de test. En effet, les cas de test qu'on génère sont destinés à détecter les fautes recensées dans le modèle de fautes. Ceci nous permet de comparer les méthodes de génération de tests selon leurs capacités à couvrir le modèle de fautes.

Dans le chapitre suivant, nous présenterons une méthode de génération de tests temporisés assurant une couverture complète de toutes les fautes étudiées dans ce chapitre.

Chapitre 7

Génération de cas de test à partir d'un AEST

Dans ce chapitre, nous présentons une méthode de génération de cas de test à partir d'un AEST et nous évaluons sa couverture de fautes par rapport au modèle de fautes temporelles introduit dans le chapitre précédent. La méthode que nous proposons est basée sur la caractérisation des états. Avant de voir plus en détail la génération de cas de test à partir d'un AEST, nous présentons d'abord les hypothèses de test qui doivent être satisfaites par l'IST, l'architecture de test utilisée et la relation de conformité adoptée.

7.1 Les hypothèses de test

Nous rappelons que les hypothèses de test ont été introduites pour simplifier le test des systèmes. En effet, le nombre d'implantations d'un système, si simple soit-il, est infini. Les hypothèses de test sont utilisées pour réduire le nombre des implantations possibles à considérer dans le test d'un système. Dans notre cas, certaines hypothèses sont nécessaires pour l'existence de l'ensemble de caractérisation pour la génération des cas de test alors que d'autres sont usuelles et permettent d'augmenter

[1]Les résultats de ce chapitre ont été publiés dans [ENDKE98, ENKD99a, ENDK01].

la couverture de fautes de notre méthodologie. Les hypothèses que nous faisons sur l'implantation à tester sont les suivantes :

- L'IST est modélisée par un AEST. Pour définir une relation de conformité entre l'implantation et la spécification, il faut que tous les deux soient données dans le même formalisme. Puisque la spécification est un AEST, nous n'avons pas le choix de supposer que l'IST est, elle aussi, un AEST.
- L'IST est déterministe. À partir de n'importe quel emplacement de l'AEST, on ne peut pas avoir deux (ou plus) transitions sortantes dont les contraintes temporelles sont simultanément satisfaites. Autrement dit, à tout moment, on ne peut avoir qu'au plus une seule transition tirable.
- L'IST utilise un nombre minimal d'horloges [DY96]. Cette hypothèse est nécessaire pour réduire le nombre de régions d'horloges dans l'implantation.
- Le domaine temporel de toute horloge de l'implantation est inclus dans le domaine temporel de l'horloge qui lui correspond dans la spécification. Cette hypothèse nous garantit que le nombre de régions d'horloges dans l'implantation est au plus égal à celui de la spécification.
- L'IST a le même alphabet que la spécification.
- L'IST a le même nombre d'emplacements que la spécification. Combinée avec les hypothèses 3 et 4, cette hypothèse garantit que le nombre d'états de l'IST est inférieur à celui de la spécification.
- L'IST est complètement spécifiée. Dans chaque état, l'implantation accepte toutes les entrées ou n'en accepte aucune.
- L'IST satisfait l'hypothèse d'équité. En soumettant une séquence d'entrées un nombre fini de fois à l'implantation, il est possible de couvrir tous les chemins d'exécution possibles pour cette séquence.

7.2 Architecture de test

Le graphe de régions définit la sémantique des AESTs dans le sens où il spécifie quand une transition est tirable et quelles sont les différentes traces du système. Cependant, il est très difficile à tester car il n'explicite pas comment traiter les variables d'horloges. Pour cela, nous proposons l'architecture de test [ENDE97, KAD+00] de la Figure 7.1 pour tester une implantation d'un système temps réel modélisé par un AEST. Dans cette architecture, l'implantation est composée de deux parties : la *partie contrôle* et la *partie horloge.*

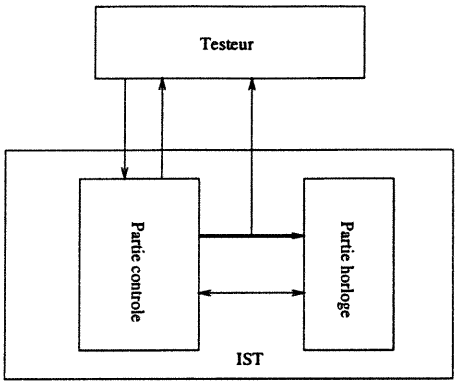

FIG. 7.1: L'architecture de test.

La partie contrôle modélise la communication du système avec son environnement en recevant les entrées et en produisant les sorties correspondantes. La partie horloge manipule les variables d'horloges utilisées dans la spécification du système. Chaque variable donne lieu à un processus chargé d'effectuer des opérations sur cette variable. La communication entre les deux parties (contrôle et horloge) de l'implantation est synchrone et est assurée par l'échange de certains signaux internes à savoir : *PleaseValue*, *GetValue* et *ResetClock.* Quand la partie contrôle reçoit une entrée de l'environnement, elle vérifie si oui ou non sa contrainte est satisfaite. Pour ce faire, elle envoie le signal *PleaseValue* aux processus d'horloges concernés pour demander les valeurs de leurs horloges. Après la réception de ce signal, chaque processus horloge calcule la valeur courante de son horloge et la communique à la partie contrôle via

le signal *GetValue*. Pour remettre à zéro une horloge, la partie contrôle envoie le signal *ResetClock* au processus de l'horloge en question. Dans la Figure 7.1, nous avons explicitement distingué le canal de *ResetClock* du reste des canaux en le mettant en gras parce que *ResetClock* est le seul signal interne qu'on veut observer.

Nous pensons que notre modèle représente d'une manière simple et efficace les systèmes temps réel à implanter et à tester. Il correspond bien au *test boîte grise* dans lequel on suppose que certaines parties de l'implantation sont connues et accessibles au testeur. De plus, notre modèle est très utilisé dans le matériel où la composante horloge est séparée du reste des composantes. L'avantage principal de cette architecture est qu'elle permet de rendre explicite la remise à zéro des horloges permettant ainsi son observation par le testeur. Ceci aide énormément à assurer une couverture complète de fautes (voir section 7.5). Sans ce modèle de test, certaines fautes peuvent échapper aux cas de test générés par notre approche.

7.3 Relation de conformité

Le point de départ du test de conformité est la définition d'une relation de conformité entre l'IST et la spécification. Une relation de conformité définit ce que signifie "une implantation est conforme à sa spécification". Plusieurs relations ont été introduites et utilisées pour la génération de cas de test à partir des MEFs et des STEs [FBK+91, LBP94, BSS86]. Elles supposent, toutes, que l'environnement se comporte correctement en respectant la spécification de référence. Le verdict sur la conformité de l'IST à sa spécification est donné en observant les réactions de l'implantation (c'est-à-dire, les sorties) suite aux entrées qui lui sont soumises. Cependant, pour tester une implantation d'un système temps réel, on doit vérifier si oui ou non l'IST répond par les sorties attendues dans les intervalles de temps permis, à la suite de sa stimulation par des entrées.

Dans notre méthodologie, nous définissons la relation de conformité entre l'IST

et la spécification comme une relation de *bisimulation temporelle modulo k* (k est un nombre rationnel strictement positif) entre deux AESTs. La définition de cette relation est comme suit.

Définition 7.1 *Bisimulation temporelle modulo k*
Soit k un nombre rationnel strictement positif. Deux états S_1 et S_2 sont dits temporellement bisimilaires modulo k (ou simplement k-temporellement bisimilaires), et on note $S_1 \sim_k S_2$, si et seulement si :

- *Pour toute action $\{?,!\}a$,*
  - *Si $S_1 \xrightarrow{\{?,!\}a} S_1'$ alors $S_2 \xrightarrow{\{?,!\}a} S_2'$ pour un certain état S_2' tel que $S_1' \sim_k S_2'$.*
  - *Si $S_2 \xrightarrow{\{?,!\}a} S_2'$ alors $S_1 \xrightarrow{\{?,!\}a} S_1'$ pour un certain état S_1' tel que $S_1' \sim_k S_2'$.*
- *Pour un délai $d = k$ correspondant à un écoulement du temps de k unités,*
  - *Si $S_1 \xrightarrow{d} S_1'$ alors $S_2 \xrightarrow{d} S_2'$ pour un certain état S_2' tel que $S_1' \sim_k S_2'$.*
  - *Si $S_2 \xrightarrow{d} S_2'$ alors $S_1 \xrightarrow{d} S_1'$ pour un certain état S_1' tel que $S_1' \sim_k S_2'$.*

Deux AESTs sont dits k-temporellement bisimilaires si et seulement si leurs états initiaux sont k-temporellement bisimilaires.

La bisimulation temporelle modulo k est une variante de la bisimulation insensible au temps [Kan95] et représente une relaxation de la bisimulation temporelle [Yi90, Čer92b]. La première condition dans la définition signifie que chaque transition explicite (sur une entrée ou une sortie) doit être appariée par une transition explicite sur la même action. Cependant, la deuxième condition ne vérifie les transitions temporelles que pour des délais rationnels de k unités de temps. Autrement dit, on ne cherche pas à vérifier tout l'espace temporelle mais seulement certaines points discrets de ce dernier.

Le nombre k dans la définition 7.1 est un paramètre qui peut varier dans un ensemble bien défini (voir section 7.4.1). Il représente, en fait, la granularité d'échantillonnage du graphe de régions que nous allons étudier dans la section suivante.

7.4 Génération des cas de test temporisés

Nous présentons, dans cette section, notre approche de génération de tests pour un système temps réel modélisé par un seul AEST. Tout comme les méthodes W, Wp et Wp généralisée, la méthodologie que nous proposons se base sur la technique de caractérisation des états et utilise une MEF minimale. La minimisation utilisée est un peu différente de la minimisation classique des MEFs car elle doit prendre en considération la sémantique du temps.

D'une manière générale, notre approche de génération de tests temporisés consiste en quatre étapes principales comme le montre la Figure 7.2 :

- L'échantillonnage du graphe de régions.
- La transformation de l'automate résultat en une machine à états finis.
- La minimisation de la MEF résultante de la transformation précédente.
- L'adaptation de la méthode Wp généralisée.

7.4.1 Échantillonnage du graphe de régions

L'échantillonnage est une technique introduite dans [LY93] pour étudier la vérification des systèmes temps réel. L'objectif était de vérifier que certaines propriétés désirables du système sont satisfaites par sa spécification. Dans notre cas, nous utilisons l'échantillonnage pour générer des cas de test temporisés afin de tester une implantation d'un système temps réel par rapport à sa spécification de référence.

En échantillonnant un graphe de régions, nous construisons un sous automate facilement testable, appelé *automate de grille.* Puisque le graphe des régions est formé des emplacements et des régions de son AEST, l'idée derrière la construction de l'automate de grille est de représenter chaque région d'horloges par un ensemble fini d'interprétations d'horloges, appelés *représentants* de la région en question. Cette technique d'échantillonnage est fondée sur la notion de points de grille dont la définition

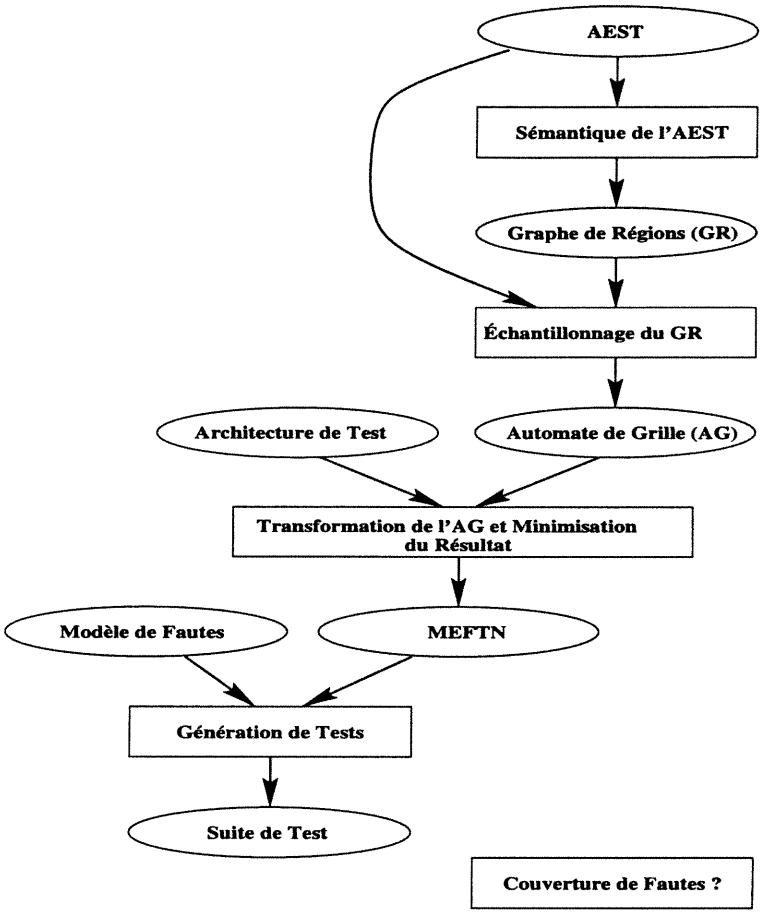

FIG. 7.2: Un aperçu général du test d'un AEST.

est comme suit.

Définition 7.2 *Points de grille*

Soit p un entier naturel strictement positif ($p \in \mathbf{N}\backslash\{0\}$). L'ensemble de points de granularité $\frac{1}{p}$ est défini par $\mathbf{N_p} = \{\frac{m}{p} | m \in \mathbf{N}\}$. Nous étendons cette définition à tout vecteur de n éléments pour définir les points de grille de granularité $\frac{1}{p}$ par l'ensemble $\mathbf{N_p^n} = \{(r_1, r_2, ..., r_n) | 1 \leq i \leq n, r_i \in \mathbf{N_p}\}$.

L'ensemble de représentants d'une région d'horloges est déterminé à partir de l'ensemble de points de grille. En fait, tous les représentants sont des éléments de l'ensemble de points de grille $\mathbf{N_p^n}$. Nous pouvons montrer que la granularité $\frac{1}{(n+1)}$ est suffisante pour déterminer les représentants de toutes les régions d'horloges d'un AEST à n horloges [LY93].

Proposition 7.1 *Représentants des régions d'horloges*

Soit $A = (I_A, O_A, L_A, l_A^0, C_A, T_A)$ un AEST avec n horloges ($|C_A| = n$). Pour toute interprétation d'horloges v de A, il existe $r \in \mathbf{N_{n+1}^n}$ tel que $v \sim r$.

Preuve

Rappelons que deux interprétations d'horloges v et v' sont équivalentes (c'est-à-dire, $v \sim v'$) si et seulement si les parties entières de leurs horloges sont les mêmes et leurs parties fractionnaires sont ordonnées de la même façon dans v et v' (voir définition 5.2).

Pour satisfaire la première condition de la définition de la relation $\sim$, nous posons $\forall x \in C, \lfloor v(x) \rfloor = \lfloor r(x) \rfloor$ (la partie entière de chaque horloge est la même dans v et r).

Pour la deuxième et la troisième condition de $\sim$, nous construisons la partie fractionnaire de chaque horloge dans r de la manière suivante :

D'abord, on ordonne les parties fractionnaires des horloges dans v : Soit $\{p_1, p_2, ..., p_n\}$ une permutation de $\{1, 2, ..., n\}$ tel que : $fract(x_{p_1}) \leq fract(x_{p_2}) \leq ... \leq fract(x_{p_n})$.

Ensuite, on pose :

$$Fract(r_{p_1}) = \begin{cases} 0 & \text{si } Fract(v_{p_1}) = 0 \\ \frac{1}{(n+1)} & sinon \end{cases}$$

Pour $1 \leq i < n$,

$$Fract(r_{p_{i+1}}) = \begin{cases} Fract(r_{p_i}) & \text{si } Fract(v_{p_i}) = Fract(v_{p_{i+1}}) \\ Fract(r_{p_i}) + \frac{1}{(n+1)} & sinon \end{cases}$$

D'après cette construction, il est clair que $v \sim r$. □

Comme une région d'horloges est une classe d'équivalence d'interprétations d'horloges induite par la relation $\sim$ et qu'à chacune de ces interprétations d'horloges correspond un point $r \in \mathbf{N^n_{n+1}}$, on garantit que chaque région d'horloges donne lieu à au moins un représentant $r \in \mathbf{N^n_{n+1}}$. Cependant, pour représenter toutes les régions d'horloges accessibles par des transitions de délai, nous devons utiliser une grille de granularité $\frac{1}{(n+2)}$ ou moins.

Proposition 7.2 *Transitions de délai et échantillonnage*
Soit $A = (I_A, O_A, L_A, l^0_A, C_A, T_A)$ *un AEST avec n horloges* ($|C_A| = n$). *Pour tout* $r \in \mathbf{N^n_{n+2}}$ *et tout délai* $d \in \mathbf{R^{>0}}$, *il existe* $r' \in \mathbf{N^n_{n+2}}$ *et* $g \in \mathbf{N_{n+2}}$ *tel que* $r \sim r'$ *et* $r + d \sim r' + g$.

Preuve

Pour satisfaire la première condition de la définition de la relation $\sim$, nous posons $\forall x \in C, \lfloor r'(x) \rfloor = \lfloor r(x) \rfloor$.

Pour la deuxième et la troisième condition de $\sim$, nous construisons la partie fractionnaire de chaque horloge dans r' comme suit :

Supposons que le délai d est strictement inférieur à 1. Soit $\{p_1, p_2, ..., p_n\}$ une permutation de $\{1, 2, ..., n\}$ tel que : $fract(x_{p_1}) \leq fract(x_{p_2}) \leq ... \leq fract(x_{p_n})$. Soit $j \in \{1, 2, ..., n\}$ le plus grand indice tel que $fract(x_{p_j} + d) = fract(x_{p_2}) + d$ (la partie entière de toute horloge ayant un indice inférieur ou égal à j ne change pas après d unités de temps). Si les parties entières de toutes les horloges ne changent pas après

d unités de temps, on prend $j = n$. Alors, les parties fractionnaires des horloges dans $r + d$ sont ordonnées de la façon suivante : $fract(x_{p_{j+1}} + d) \leq fract(x_{p_{j+2}} + d) \leq ... \leq fract(x_{p_n}) \leq fract(x_{p_1} + d) \leq fract(x_{p_2} + d) \leq ... \leq fract(x_{p_j} + d)$.
Pour définir la partie fractionnaire de r', on distingue les cas suivants :

- Si $j = n$ (c'est-à-dire, $r + d \sim r$), on prend $r' = r$ et $g = 0$.
- Si $fract(x_{p_{j+1}} + d) = 0$ (c'est-à-dire, l'horloge $x_{p_{j+1}}$ change sa partie entière et le délai d est un multiple de $\frac{1}{(n+2)}$), on prend $r' = r$ et $g = d$.
- Dans les autres cas,

$$Fract(r'_{p_1}) = \begin{cases} 0 & \text{si } Fract(r_{p_1}) = 0 \\ \frac{1}{(n+2)} & sinon \end{cases}$$

Pour $1 \leq i \leq j$,

$$Fract(r'_{p_{i+1}}) = \begin{cases} Fract(r'_{p_i}) & \text{si } Fract(r_{p_i}) = Fract(r_{p_{i+1}}) \\ Fract(r'_{p_i}) + \frac{1}{(n+2)} & sinon \end{cases}$$

$$Fract(r'_{p_{j+1}}) = Fract(r'_{p_{j+1}}) + \frac{1}{(n+2)}$$

Pour $j + 1 \leq i < n$,

$$Fract(r'_{p_{i+1}}) = \begin{cases} Fract(r'_{p_i}) & \text{si } Fract(r_{p_i}) = Fract(r_{p_{i+1}}) \\ Fract(r'_{p_i}) + \frac{1}{(n+2)} & sinon \end{cases}$$

Si on prend $g = 1 - (fract(r'_{p_j}) + \frac{1}{(n+2)})$, on aura $g \in \mathbf{N_{n+2}}$, $r \sim r'$ et $r + d \sim r' + g$. Dans le cas où le délai d est supérieur ou égal à 1, on suit la démarche précédente en utilisant le délai d' ($d' = d - \lfloor d \rfloor$) et en ajoutant à la fin la valeur $\lfloor d \rfloor$ à g. □

La granularité des points de grille représente le pas par lequel le système est autorisé de passer d'une région d'horloges à une autre afin qu'il puisse exécuter une transition explicite (sur une entrée ou une sortie). Nous donnons maintenant la définition formelle de l'automate de grille. Celle-ci ressemble à celle du graphe de régions dans laquelle les transitions de délai sont réduites à la granularité d'échantillonnage.

Définition 7.3 *Automate de grille*

Soit $A = (I_A, O_A, L_A, l_A^0, C_A, T_A)$ *un AEST. L'automate de grille correspondant à cet AEST est un système de transitions étiquetées* $G = (\Sigma_G, S_G, s_G^0, T_G)$, *où :*

- $\Sigma_G = I_A \cup O_A \cup \{g\}$ *est l'ensemble d'alphabets, où* g *est la granularité d'échantillonnage.*
- $S_G \subset S_A$ *est l'ensemble des états.*
- $s_G^0 = (l_A^0, v_0)$ *est l'état initial, où* v_0 *est l'interprétation d'horloges qui remet toutes les horloges à zéro.*
- T_G *est l'ensemble de transitions tel que :*
  - T_G *contient une transition* $(l_i, v_i) \xrightarrow{\{!,?\}a} (l_j, [R := 0]v_i)$ *pour toute transition* $l_i \xrightarrow{\{!,?\}a,R,G} l_j$ *dans* T_A *tel que* $v_i \models G$.
  - T_G *contient une transition* $(l_i, v_i) \xrightarrow{g} (l_i, v_i + g)$ *pour tout état* $(l_i, v_i) \in S_G$ *tel que* $\exists x \in C_A$ *et* $v(x) + g \neq \infty$.
  - T_G *contient une transition* $(l_i, v_i) \xrightarrow{g} (l_i, v_i)$ *pour tout état* $s \in S_G$ *tel que* $\forall x \in C_A,\ v(x) + g = \infty$.

La construction de l'automate de grille donne lieu à l'extension explicite de l'alphabet de l'automate initial par une action de délai d'une valeur $\frac{1}{(n+2)}$ si $n \geq 2$, ou $\frac{1}{2}$ sinon. C'est cette valeur de délai (la granularité d'échantillonnage) qu'on utilise pour instancier le paramètre k dans la définition 7.1 afin de définir la relation de conformité entre l'implantation sous test et la spécification de référence données, toutes les deux, sous forme d'AESTs. On peut facilement remarquer que deux AESTs sont k-temporellement bisimilaires si et seulement si leurs automates de grille sont traces-équivalents.

Pour mieux comprendre l'idée d'échantillonnage, nous prenons comme exemple l'automate de la Figure 7.3. Cet automate décrit un commutateur simple qui reçoit une entrée $?In$ et produit la sortie $!Out$ au plus tard une unité de temps après. Ici, on utilise une seule horloge, x, pour spécifier la contrainte temporelle entre la réception de l'entrée $?In$ et la production de la sortie $!Out$. Donc, nous utilisons

une granularité de $\frac{1}{2}$ pour calculer les points de grille. L'ensemble des représentants est alors $\{(0), (\frac{1}{2}), (1), (\infty)\}$. Nous obtenons ainsi l'automate de grille de la Figure 7.4.

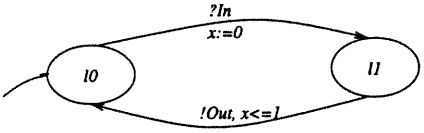

FIG. 7.3: Un AEST à une seule horloge.

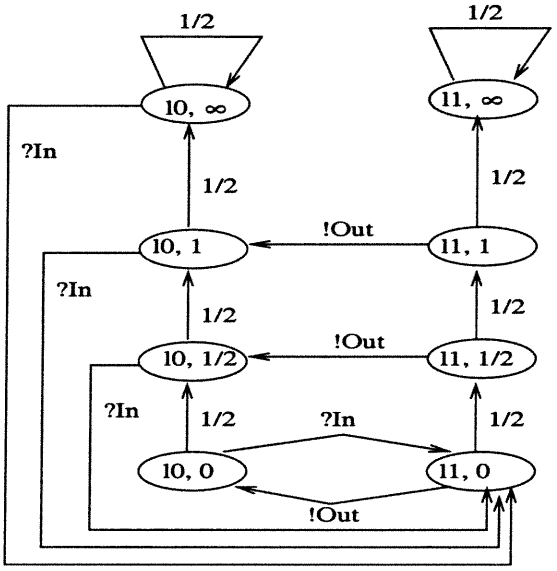

FIG. 7.4: L'automate de grille de l'AEST de la Figure 7.3.

L'algorithme d'échantillonnage d'un graphe de régions est décrit dans la Figure 7.5. Il a comme entrée un automate à entrées sorties temporisées à n horloges et produit comme résultat un automate de grille conformément à la définition 7.3.

Pour construire l'automate de grille, notre algorithme procède comme suit. Dans la première étape, on dérive la granularité maximale à utiliser pour l'échantillonnage.

La granularité ne dépend que du nombre d'horloges, n, dans l'AEST ; elle est de $\frac{1}{(n+2)}$ si $n \geq 2$ et $\frac{1}{2}$ sinon. Dans la deuxième étape, on crée l'état initial du système. Celui-ci, rappelons le, est formé de l'emplacement initial de l'AEST et de l'interprétation d'horloges qui met toutes les horloges à zéro. Dans la troisième étape, on choisit un état non encore traité (l, v) et on crée une transition $((l, v), \{?, !\}a, (l', [R := 0]v))$ pour toute transition $(l, l', \{?, !\}a, R, G)$ dans l'AEST tel que v satisfait G. De même, on crée une transition de délai $((l, v), \frac{1}{(n+2)}, (l', v + \frac{1}{(n+2)}))$ ou $((l, v), \frac{1}{2}, (l', v + \frac{1}{2}))$, dépendamment de la granularité utilisée. Dans cette étape, nous rajoutons aussi les états d'arrivée des transitions crées à l'ensemble d'états de l'automate s'ils n'y figurent pas. Nous répétons l'étape 3 jusqu'à ce que tous les états soient traités.

Entrées : - Un AEST avec k horloges $A = (I, O, L, l^0, C, T)$.
Sorties : - Un automate de grille.
Étape0 : Calcul de la granularité à utiliser :
  granul $\leftarrow$ si $k = 1$ alors $\frac{1}{2}$ sinon $1/(k + 2)$.
Étape1 : Initialisation des ensembles à utiliser :
  $R_S \leftarrow (l_0, v_0(x) = 0) \forall x \in C$ (États Accessibles)
  $H_S \leftarrow \emptyset$ (États Traités)
Étape2 : Construction de l'automate de grille :
  Tant Que $R_S \backslash H_S \neq \emptyset$ Faire
    Choisir un état $s = (l, v)$ dans $R_S \backslash H_S$
    $H_S \leftarrow H_S \cup \{s\}$
    Pour chaque $t = (l, l', \{!, ?\}a, R, G) \in T$ Faire
      Si $v \models G$ alors
        Ajouter $((l, v), \{!, ?\}a, (l', [R := 0]v))$ s'il n'existe pas dans l'ensemble des transitions
        $R_S \leftarrow R_S \cup \{(l', [R := 0]v)\}$ s'il n'existe pas dans R_S
      FinSi
    FinPour
    Ajouter $((l, v), granul, (l, (v + granul)))$ s'il n'existe pas dans l'ensemble des transitions
    $R_S \leftarrow R_S \cup \{(l, (v + granul))\}$ s'il n'existe pas dans R_S
  FinTantQue

FIG. 7.5: L'algorithme d'échantillonnage.

L'algorithme d'échantillonnage présenté à la Figure 7.5 construit correctement l'automate de grille de tout AEST. En fait, cet algorithme est une traduction di-

recte de la définition 7.3. Pour prouver sa validité, il suffit de s'assurer de différentes caractéristiques de l'automate de grille énoncées dans la définition 7.3 :

- L'ensemble d'alphabets.
- L'ensemble des états.
- L'ensemble des transitions.

Proposition 7.3 Validité de l'algorithme d'échantillonnage

Soit $A = (I_A, O_A, L_A, l_A^0, C_A, T_A)$ *un AEST. Notre algorithme d'échantillonnage construit correctement l'automate de grille* $G = (\Sigma_G, S_G, s_G^0, T_G)$ *de* A.

Preuve

Soit $A = (I, O, L, l_0, C, T)$ un AEST. Montrons par construction que l'automate de grille $G = (\Sigma_G, S_G, s_G^0, T_G)$ obtenu en appliquant l'algorithme 7.5 est conforme à la définition 7.3.

- La granularité d'échantillonnage nécessaire à la construction de l'automate de grille de A est garantie par la proposition 7.2. Cette granularité est calculée à l'étape0 de l'algorithme.

- L'algorithme construit l'état initial de l'automate de grille de A en le calculant à l'étape1.

- Pour l'ensemble de transitions de l'automate de grille, l'algorithme les construit à l'étape2 en parcourant toutes les transitions de l'AEST initial. Toute transition dont la contrainte temporelle est satisfaite par un état de l'automate de grille donne lieu à une transition dans l'automate de grille. L'état d'arrivée de la nouvelle transition est ajouté à l'ensemble des états de l'automate de grille. L'action de la transition est automatiquement ajoutée à l'ensemble de l'alphabets de l'automate de grille. De cette façon, on retrouve dans l'automate de grille toutes les transitions possibles de l'AEST. Les transitions de délai dans l'automate de grille sont crées à l'étape2. Ceci garantit que chaque état de l'automate de grille ait une transition sortante étiquetée par la granularité d'échantillonnage.

La condition de la boucle *Tant Que* de l'étape2 de l'algorithme ne devienne fausse que lorsque tous les états de l'automate de grille soient traîtés. On en déduit qu'à la

fin de l'algorithme :

- L'ensemble d'alphabets de l'automate de grille est égal à $(I \cup O \cup \{g\})$, où g est la granularité d'échantillonnage.
- L'ensemble des états de l'automate de grille est formé de tous les états accessibles à partir de l'état initial par des transitions explicites (sur des entrées sorties) et des transitions de délai d'une durée égale à la granularité d'échantillonnage.
- Chaque état de l'automate de grille a une transition sortante étiquetée par la granularité d'échantillonnage.

□

Nous notons que notre algorithme construit un automate de grille similaire à celui défini dans [SVD01] sauf que, ici, nous utilisons une grande granularité nous permettant, plus tard, de réduire le nombre de cas de test générés pour le système qu'on veut tester. Nous signalons aussi que la complexité de notre algorithme est exponentielle en termes du nombre d'horloges et des constantes utilisés dans l'AEST de la spécification. Cependant, cette complexité n'enlève rien à notre algorithme puisqu'elle est inhérente à toutes les méthodes de test et de vérification basées sur le graphe des régions.

7.4.2 Transformation de l'automate de grille en une MEF

Dans le test des systèmes temps réel, la contrôlabilité du temps auquel une implantation produit une sortie est difficile ou presque impossible sans recourir à l'estampillage des sorties par l'implantation. Ceci est dû principalement au fait que le temps n'est pas sous un contrôle direct du testeur. Pour résoudre ce problème, nous ne nous intéresserons pas au moment exact de la production d'une sortie mais nous vérifions si oui ou non elle a eu lieu dans l'intervalle de temps défini dans la spécification du système. Autrement dit, nous voulons vérifier uniquement si les contraintes des sorties sont satisfaites ou pas, sans être obligés de connaître le moment exact auquel elles se produisent. Pour ce faire, nous transformons l'automate de grille en une ma-

chine à états finis non déterministe, appelée *Machine à États Finis Temporisée Non déterministe (MEFTN)*. Le non déterminisme est principalement dû aux transitions sur les sorties qui sont imprévisibles. Dans la transformation que nous effectuons, l'action correspondant à l'écoulement du temps est vue comme une entrée. Ceci dit que, lors du test d'une implantation, les testeurs (le testeur inférieur et le testeur supérieur) vérifient régulièrement (chaque g unités de temps où g est la granularité d'échantillonnage) la queue de sorties de l'implantation sous test. Pour couvrir tout l'espace temporel des sorties par la granularité d'échantillonnage, nous supposons que les contraintes temporelles sur les sorties dans l'AEST sont de la forme $x \leq m$, $x = m$, $x \geq m$ et $x > m$.

L'algorithme de transformation de l'automate de grille en une MEFTN est donné à la Figure 7.6.

Entrée : - Un Automate de Grille G.
Sortie : - Une MEFTN M.
Traitement :
Pour chaque état s_i de G Faire
Pour chaque transition $s_i \xrightarrow{?a} s_j$ de G Faire
Pour chaque transition $s_j \xrightarrow{\{?,!\}b} s_k = (l_k, v_k)$ de G Faire
Si b est une sortie alors
Ajouter s_i, s_j et s_k à l'ensemble des états de M
Ajouter $s_i \xrightarrow{a/b} s_k$ et $s_i \xrightarrow{a/-} s_j$ à l'ensemble des transitions de M
Ajouter les sorties correspondant aux remises à zéro des horloges entre les états de ces transitions
Sinon
Ajouter s_i, s_j et s_k à l'ensemble des états de M
Ajouter $s_i \xrightarrow{a/-} s_j$ à l'ensemble des transitions de M
Ajouter les sorties correspondant aux remises à zéro des horloges entre les états de cette transition
FinSi
FinPour
FinPour
FinPour

FIG. 7.6: L'algorithme de transformation de l'automate de grille.

L'algorithme de transformation d'un automate de grille en une MEFTN est basé sur les deux schémas de la Figure 7.7. L'idée de cette transformation est de coupler chaque sortie avec l'entrée ou l'action de délai qui la précède. Si on a une transition sur une entrée $?a$ d'un état s_i à un état s_j, suivie d'une transition de sortie $!b$ de l'état s_j à l'état s_k, on crée dans la MEFTN deux transitions $s_i \xrightarrow{a/b} s_k$ et $s_i \xrightarrow{a/-} s_j$. Par contre, si on a une transition sur une entrée $?a$ d'un état s_i à un état s_j, suivie d'aucune transition de sortie $!b$ de l'état s_j à l'état s_k, on crée crée dans la MEFTN une seule transition $s_i \xrightarrow{a/-} s_j$. Dans l'algorithme, une transition de délai est traîtée comme une transition sur une entrée. Il est clair de schémas de la Figure 7.7 et de l'algorithme donné à la Figure 7.6 que la transformation de l'automate de grille en une MEFTN se fait correctement.

La Figure 7.8 montre la MEFTN correspondante à l'automate de grille de la Figure 7.4.

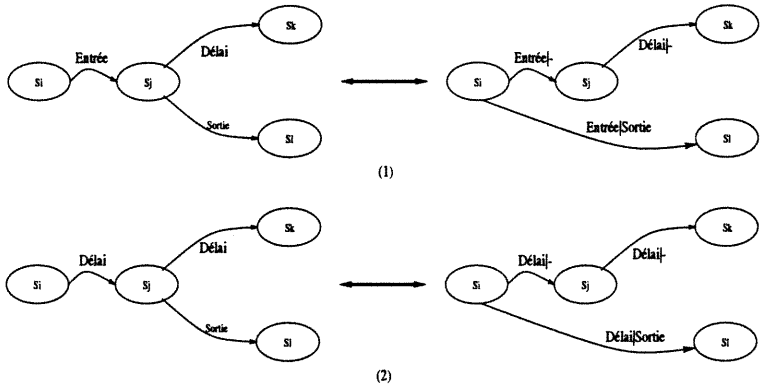

FIG. 7.7: Les schémas de transformation

Nous signalons que la transformation de la Figure 7.7 préserve le comportement exprimé par l'automate de grille assurant ainsi l'équivalence entre l'automate de grille et la MEFTN résultante de la transformation. Nous mentionnons aussi que la MEFTN est non déterministe observable dans le sens où le testeur peut connaître l'état de l'implantation après l'exécution de chaque transition. De plus, la relation de bisimulation temporelle modulo k entre deux AESTs se réduit à une équivalence de traces

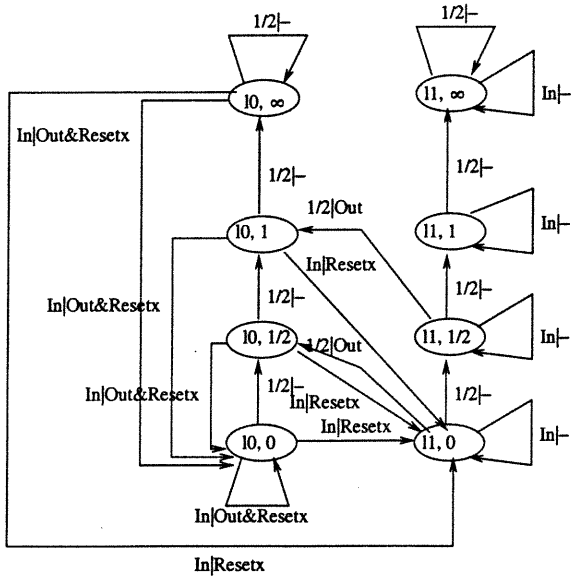

FIG. 7.8: La MEFTN correspondant à l'automate de grille de la Figure 7.4.

entre leurs MEFTNs. Deux MEFTNs sont traces-équivalentes si et seulement si les ensembles de traces de leurs états initiaux sont identiques.

Proposition 7.4 Bisimulation temporelle modulo k versus équivalence de traces

Deux AESTs sont bisimilaires temporellement modulo k si et seulement si leurs MEFTNs sont traces-équivalentes.

Preuve

• Soient deux AESTs k-temporellement bisimilaires $A1$ et $A2$. Montrons que leurs MEFTNs, respectivement $M1$ et $M2$, sont traces-équivalentes. $A1$ et $A2$ sont k-temporellement bisimilaires signifie que leurs automates de grille respectifs sont traces-équivalents. Les MEFTNs $M1$ et $M2$ sont obtenues à partir des automates de grille de $A1$ et $A2$ en utilisant les transformations de la Figure 7.7. Par construction, ces transformations préservent le comportement (l'ensemble de traces) de l'automate de grille. Alors, les MEFTNs $M1$ et $M2$ sont traces-équivalentes.

• Supposons maintenant qu'on a deux MEFTNs $M1$ et $M2$ qui sont traces-équivalentes et montrons que leurs AESTs, respectivement $A1$ et $A2$, sont k-temporellement bisimilaires. Supposons le contraire ; c'est-à-dire, les deux AESTs $A1$ et $A2$ ne sont pas k-temporellement bisimilaires. Sans perte de généralité, on suppose que les sorties des automates sont attendues à des instants entiers. D'après l'hypothèse 2 dans la section 7.1, les automates $A1$ et $A2$ sont déterministes.
$A1$ et $A2$ ne sont pas k-temporellement bisimilaires signifie que leurs états initiaux respectifs $S0$ et $S0'$ ne sont pas k-temporellement bisimilaires. Ceci dit qu'il existe deux état $S1$ et $S2$ dans $A1$, un autre état $S1'$ dans $A2$, une trace σ possiblement nulle et une action a tel que : $S0 \xrightarrow{\sigma} S1, S0' \xrightarrow{\sigma} S1', S1 \xrightarrow{a} S2$ et $S1' \overset{a}{\not\longrightarrow}$ ($S1$ et $S1'$ ne sont pas k-temporellement bisimilaires). Regardons les différents cas possibles de a.

- a ne peut être une action de délai car le temps est en progression continue et n'est pas sous le contrôle du système.
- a ne peut être une action d'entrée car les automates qu'on traite sont supposés être complètement spécifiés.
- a ne peut être alors qu'une action de sortie. Elle est donc observable. En suivant les transformations de la Figure 7.7, on trouve au niveau des MEFTNs $M1$ et $M2$ deux états $S \in M1$ et $S' \in M2$ correspondant respectivement aux $S1$ et $S1'$ tel que : $S \xrightarrow{i/a}$ et $S' \xrightarrow{i/-}$, où i est une action d'entrée ou de délai et "$-$" est la sortie nulle. On en déduit que les deux MEFTNs $M1$ et $M2$ ne sont pas traces-équivalentes. Ceci est en contradiction avec notre hypothèse de départ. Par conséquent, les deux AESTs $A1$ et $A2$ sont k-temporellement bisimilaires.□

Par ailleurs, la MEFTN que nous venons de construire a la caractéristique d'avoir deux catégories d'états :

- Les états connectés par des transitions de délai : ces états ont le même premier composant (c'est-à-dire, l'emplacement) et une transition entrante étiquetée par l'entrée $\frac{1}{(n+2)}$ ou $\frac{1}{2}$, dépendamment de la granularité d'échantillonnage. Ces états

ne peuvent pas être distingués par une séquence d'entrées car ils sont générés à partir du même emplacement dans l'AEST initial. Cependant, la sémantique du temps permet de distinguer ces états entre eux à condition qu'aucune horloge ne soit remise à zéro dans la dernière $\frac{1}{(n+2)}$ ou $\frac{1}{2}$ unités de temps. En effet, la sémantique du temps garantit que le système change son état en laissant le temps s'écouler de $\frac{1}{(n+2)}$ ou $\frac{1}{2}$ unités de temps. De plus, l'architecture de test que nous utilisons nous permet de savoir si une horloge est remise à zéro ou pas.

- Les états qui sont distinguables par des séquences d'entrées tout comme dans le cas des MEFs.

L'ensemble des états connectés par des transitions de délai et l'ensemble des états distinguables par des séquences d'entrées sont respectivement notés par S_T et S_I. À titre d'exemple, nous trouvons pour la Figure 7.8 : $S_I = \{(l_0, 0), (l_1, 0)\}$ et $S_T = \{\{(l_0, 0), (l_0, \frac{1}{2}), (l_0, 1), (l_0, \infty)\}, \{(l_1, 0), (l_1, \frac{1}{2}), (l_1, 1), (l_1, \infty)\}\}$.

7.4.3 Minimisation de la MEFTN

Comme nous l'avons précédemment mentionné, notre approche de génération de cas de test utilise une machine à états finis minimale. La machine à minimiser est la MEFTN obtenue suite à la transformation de l'automate de grille. La définition d'une MEFTN minimale est comme suit.

Définition 7.4 *MEFTN minimale*
Soit $M = (I_M, O_M, S_M, s^0_M, T_M)$ une MEFTN. M est dite minimale si pour tout état $s_i \in S_M$, il n'existe aucun état $s_j \in S_M$ traces-équivalent à s_i et ayant la même interprétation d'horloges que s_i et un emplacement différent de celui de s_i.

L'algorithme de minimisation d'une MEFTN est donné à la Figure 7.9. Cette minimisation est légèrement différente de la minimisation classique des MEFs car il faut prendre en considération le temps. En effet, au lieu de vérifier l'équivalence entre chaque paire d'états, nous ne considérons que les paires d'états dont les emplacements sont différents mais leurs interprétations d'horloges sont les mêmes. Autrement dit,

nous ne vérifions l'équivalence de traces que pour les états (l, v) et (l', v) tel que $l \neq l'$ (voir l'algorithme). Pour cela, nous parcourons tous les états de la MEFTN, et pour chaque paire d'états (l, v) et (l', v), nous vérifions d'une manière récursive s'ils acceptent les mêmes traces. Si un des deux états accepte une trace ne faisant pas partie des traces de l'autre, les deux états sont jugés non équivalents. Ils sont alors distinguables par au moins une séquence d'entrées. Par contre, si toutes les traces des deux états sont examinées et qu'aucune d'entre elles ne peut distinguer un état de l'autre, les deux états sont jugés équivalents. Dans ce cas, nous supprimons un de ces deux états ainsi que ses enfants (c'est-à-dire, tous les états accessibles à partir de lui). Évidemment, toutes les transitions sortantes de l'état enlevé sont supprimées et toutes les transitions y entrant (c'est-à-dire, provenant de ses parents) sont aiguillées vers l'état équivalent restant. L'algorithme de vérification de l'équivalence entre deux états d'une MEFTN est donné à la Figure 7.10.

Proposition 7.5 Validité de l'algorithme de minimisation

Soit $M = (I_M, O_M, S_M, S^0_M, T_M)$ une MEFTN. L'algorithme de la Figure 7.9 minimise correctement la machine M.

Preuve

Soit $M = (I_M, O_M, S_M, S^0_M, T_M)$ une MEFTN. Montrons par construction que la machine résultat de l'algorithme de la Figure 7.9 est une forme minimale de M conformément à la définition 7.4.

L'algorithme de la Figure 7.9 parcourt tous les états de la MEFTN en utilisant deux boucles *Pour* imbriquées permettant ainsi de comparer chaque état avec tous les autres états (on examine toutes les paires d'états possibles). Si les deux états d'une paire ont la même interprétation d'horloges et diffèrent selon leurs emplacements, on vérifie leur équivalence à l'aide de l'algorithme de la Figure 7.10. Ce dernier est celui utilisé dans la vérification de l'équivalence des états d'une MEF [Koh78]. Lorsque les états de la MEFTN sont équivalents, on supprime l'un d'eux avec ses enfants tout en aiguillant les transitions y entrant vers l'état restant.

Entrée : - Une MEFTN non minimale.
Sortie : - Une MEFTN minimale.
Traitement :
Pour chaque état $s_i = (l_i, v_i)$ de la MEFTN Faire
Pour chaque état $s_j = (l_j, v_j)$ de la MEFTN Faire
Si $((l_i \neq l_j)$ et $(v_i = v_j))$ alors
Si (ÉtatsÉquivalents(s_i, s_j) = vrai) alors
Détourner chaque transition entrant à s_j vers s_i
Supprimer l'état s_j avec leurs enfants
FinSi
FinSi
FinPour
FinPour

FIG. 7.9: L'algorithme de minimisation.

Fonction ÉtatsÉquivalents(S_i et S_j) retourne un Booléen
S'il n'y a aucune transition $S_i \xrightarrow{si_{in}|si_{out}} S_{di}$ et aucune transition $S_j \xrightarrow{sj_{in}|sj_{out}} S_{dj}$
Alors retourner Vrai
FinSi
Pour chaque transition $S_i \xrightarrow{si_{in}|si_{out}} S_{di}$ Faire
T_{Equiv} := Faux
E_{Equiv} := Faux
Pour chaque transition $S_j \xrightarrow{sj_{in}|sj_{out}} S_{dj}$ Faire
Si $si_{in} = sj_{in}$ et $si_{out} = sj_{out}$ Alors
T_{Equiv} := Vrai
Si $S_i = S_{di}$ et $S_j = S_{dj}$ Alors
E_{Equiv} := Vrai
Sinon
E_{Equiv} := ÉtatsÉquivalents(S_{di}, S_{dj})
FinSi
FinSi
FinPour
Si (T_{Equiv} = Faux) ou (E_{Equiv} = Faux) Alors
Retourner Faux
FinSi
FinPour
Retourner Vrai

FIG. 7.10: L'algorithme de vérification de l'équivalence entre deux états.

À la fin de l'algorithme, toutes les paires d'états seront traîtées et s'il y'avait des états équivalents (avec des emplacements différents et une même interprétation d'horloges), un d'eux serait supprimé. Ainsi, la MEFTN résultat sera minimale conformément à la définition 7.4. □

7.4.4 Adaptation de la méthode Wp généralisée

Le but des transformations introduites dans les sections précédentes est de ramener le test d'un AEST à celui d'une MEF avec certaines particularités. En effet, la construction de l'automate de grille nous a menés, comme nous l'avons précédemment mentionné, à l'augmentation explicite de l'alphabet de l'AEST par l'action de délai $\frac{1}{(n+2)}$ ou $\frac{1}{2}$ de façon à ce que chaque état ait une transition sortante étiquetée par cette action et qu'il puisse être complété pour le reste des actions d'entrée. De plus, nous utilisons notre architecture de test pour rendre explicite la remise à zéro à l'interface de communication entre la partie contrôle et la partie horloge du système. D'autant plus, nous avons montré comment minimiser la MEFTN résultante de toutes les modifications précédemment effectuées. Ainsi, nous pouvons facilement appliquer les méthodes de génération de tests basées sur la caractérisation des états pour dériver une suite de test pour le système temps réel qu'on veut tester.

Les méthodes de caractérisation des états génèrent les cas de test en deux phases. La première phase vérifie si tous les états de la spécification sont identifiables dans l'implantation ou pas. La deuxième phase, quant à elle, vérifie si toutes les transitions de la spécification sont correctement implantées ou pas. Généralement, la deuxième phase ne considère que les transitions qui n'ont pas été couvertes dans la première phase.

La Figure 7.11 montre l'algorithme de la *méthode Wp temporisée*. Il s'agit d'une adaptation de la méthode Wp généralisée (voir section 4.1.6) pour la génération de

cas de test temporisés à partir de l'AEST. La méthode Wp temporisée utilise plusieurs ensembles dont les définitions sont comme suit.

Définition 7.5 Couverture d'états

Un ensemble $Q \subseteq I^*$ est dit une couverture d'états d'une MEFTN M *si, pour chaque état S_i,* Q *contient une séquence d'entrées qui amène la machine* M *de son état initial S_0 à l'état S_i.*

Définition 7.6 Couverture de transitions

Un ensemble $P \subseteq I^*$ est dit une couverture de transitions d'une MEFTN M si, pour chaque transition $S_i \xrightarrow{a|b}_M S_j$, P contient deux séquences d'entrées x et $x.a$ qui amènent respectivement la machine M de son état initial S_0 aux états S_i et S_j. Nous notons que $Q \subseteq P$.

Définition 7.7 Ensemble de séquences de délai

Un ensemble $D \subseteq (\{\frac{1}{(n+2)}\} \cup I)^*$ est dit un ensemble de séquences de délai d'une MEFTN M si, pour chaque transition $S_i \xrightarrow{\frac{1}{(n+2)}|b}_M S_j$, D contient deux séquences d'entrées x et $x.\frac{1}{(n+2)}$ qui amènent respectivement la machine M de son état initial S_0 aux états S_i et S_j. Nous notons aussi que $D \subseteq P$.

Définition 7.8 Ensemble de caractérisation

Un ensemble $W \subseteq I^*$ est dit un ensemble de caractérisation d'une MEFTN si et seulement si $\forall S_i, S_j, i \neq j \Rightarrow \exists x$ tel que : $x \in Traces(S_i)$ xor $Traces(S_j)$, et $x\rceil I \in W$, où $x\rceil I$ est la projection de x sur I obtenue simplement en supprimant les actions de sorties dans x. Avec l'hypothèse de complétion, ceci signifie que l'application de $(x\rceil I)$ à S_i et S_j donne lieu à deux ensembles différents de traces.

Définition 7.9 Ensemble de préfixes

Soit V *un ensemble de séquence d'entrées. On définit l'ensemble de préfixes de* V *par :* $Pref(V) = \{t_1 \mid t_1 \neq \varepsilon \wedge t_2 \in I^* \& t_1.t_2 \in V\}$.

Définition 7.10 Ensemble d'identification des états

Soient une MEFTN et un ensemble de caractérisation W, $\{W_0, W_1, ..., W_{n-1}\}$ est dit

un ensemble d'identification des états si, pour i = 0, 1, ..., n-1, W_i est un ensemble minimal tel que : $W_i \subseteq Pref(W)$ et pour $j = 0, 1, ..., n-1, j \neq i \Rightarrow \exists x \in Traces(S_i)$ xor $Traces(S_j)$ et $x]I^* \in W_i$).

Entrées :
- Un AEST A avec k horloges et un ensemble d'alphabets $\Sigma = I \cup O$,
- Une IST M,

Sorties :
- Une suite de test pour l'IST M.

Étape0 : Construire un automate de grille de l'AEST A
avec un alphabet $\Sigma \cup \{\frac{1}{(k+2)}$ ou $\frac{1}{(k+1)}\}$ et n états, en utilisant l'algorithme de la section 7.4.1,
Étape1 : Transformer l'automate de grille en une
MEFTN selon la procédure de la section 7.1,
Étape2 : Déterminer les ensembles S_T et S_I.
Étape3 : Construire l'ensemble de couverture d'états Q qui inclut
l'ensemble de séquences de délai D,
Étape4 : Construire l'ensemble de caractérisation W et
l'ensemble d'identification des états $\{W_0, W_1, ..., W_{n-1}\}$ pour la MEFTN.
Étape5 : Construire les deux ensembles P et R tel que :
$P = Q.(I \cup \{\varepsilon\})$ et $R = P \backslash Q$.
Étape6 : Construire la suite de test temporisée $\prod$ comme suit :
$\prod = \prod 1 \cup \prod 2$, où
$\prod 1 = Q.W$,
$\prod 2 = R \oplus \{W_0, W_1, ..., W_{n-1}\}$, où

$$R \oplus \{W_0, W_1, ..., W_{n-1}\} = S_0 \xrightarrow{x} \bigcup_{\{x]I\}.W_i.} S_i \& x]I \in R$$

FIG. 7.11: La méthode Wp temporisée

Comme nous pouvons le remarquer, la méthode Wp temporisée procède en six étapes pour générer les cas de test à partir d'un AEST. Dans les étapes 1 et 2, on transforme l'AEST en une MEFTN en suivant la procédure expliquée dans les sections précédentes. Dans les trois étapes suivantes (étapes 3, 4 et 5), on construit les ensembles S_T, S_I, Q, W et $\{W_0, W_1, ..., W_{n-1}\}$ tels qu'ils sont définis précédemment. Ensuite, on dérive la couverture de transitions P, en concaténant l'ensemble Q avec l'ensemble des entrées I et l'entrée nulle ($P = (I \cup \{\varepsilon\})$). À ce niveau, on construit aussi l'ensemble R qui inclut toutes les transitions de la MEFTN à l'exception de

celles faisant partie de Q. Enfin, dans l'étape 7, on construit la suite de test qui est formée de deux sous ensembles $\prod 1$ et $\prod 2$. $\prod 1$ est la concaténation des ensembles Q et W. Quant à $\prod 2$, il est calculé en concaténant chaque séquence d'entrées $x]I$ avec l'ensemble d'identification W_i de tout état S_i, accessible à partir de l'état initial de la MEFTN par l'application de la séquence $x]I$.

L'application de la méthode Wp temporisée sur la MEFTN de la Figure 7.8 donne lieu aux résultats de la Figure 7.12.

L'algorithme de la Figure 7.11 génère des cas de test afin de tester une implantation par rapport à sa spécification donnée sous forme d'un AEST. Lorsqu'une implantation passe avec succès les cas de test générés, elle sera déclarée k-temporellement bisimilaire à sa spécification.

Théorème 7.1 Conformité d'une implantation passant avec succès le test
Toute implantation passant avec succès les cas de test générés par la méthode Wp temporisée est k-temporellement bisimilaire à sa spécification.

Preuve
La preuve est directe et se base sur la preuve de la méthode Wp généralisée et la proposition 7.4. La méthode Wp temporisée génère les cas de test à partir de la MEFTN de l'AEST de la spécification. Selon la preuve de la méthode Wp généralisée, si une implantation passe avec succès les cas de test générés, sa MEFTN est traces-équivalente à celle de sa spécification. Or, d'après la proposition 7.4, deux MEFTNs sont traces-équivalentes si et seulement si leurs AESTs sont k-temporellement bisimilaires. Donc, si une implantation passe avec succès les cas de test générés par la méthode Wp temporisée, elle est k-temporellement bisimilaire à sa spécification.□

Pour ce qui est de la détection des fautes suceptibles d'exister dans l'implantation, les cas de test générés par l'algorithme ne vérifient que le comportement de l'automate de grille dans l'implantation mais ne couvrent pas tout le graphe des régions.

Les états de la MEFTN
$s_0 = (l_0, 0), s_1 = (l_0, \frac{1}{2}), s_2 = (l_0, 1), s_3 = (l_0, \infty),$ $s_4 = (l_1, 0), s_5 = (l_1, \frac{1}{2}), s_6 = (l_1, 1), s_7 = (l_1, \infty)$
L'ensemble S_T
$\{\{(l_0, 0), (l_0, \frac{1}{2}), (l_0, 1), (l_0, \infty)\}, \{(l_1, 0), (l_1, \frac{1}{2}), (l_1, 1), (l_1, \infty)\}\}$
L'ensemble S_I
$\{(l_0, 0), (l_1, 0)\}$
L'ensemble P
$\{\varepsilon, \frac{1}{2}, \frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}, In, \frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.$ $In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}\}$
L'ensemble Q
$\{\varepsilon, \frac{1}{2}, \frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}\}$
L'ensemble D
$\{\frac{1}{2}, \frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}\}$
L'ensemble R
$\{In, \frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.In, \frac{1}{2}.In.\frac{1}{2}.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.$ $\frac{1}{2}.\frac{1}{2}.\frac{1}{2}\}$
L'ensemble W
$\{\frac{1}{2}, In\}$
Les ensembles W_i
$W_0 = W_1 = W_2 = W_3 = W_6 = W_7 = \{In\}$ et $W_4 = W_5 = \{\frac{1}{2}\}$
Les case de test $\prod 1$:
$\{\frac{1}{2}, In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.In, \frac{1}{2}.In.\frac{1}{2}.In, \frac{1}{2}.In,$ $\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In\}$
Les cas de test $\prod 2$:
$\{In.In, In.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.In.In, \frac{1}{2}.\frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In.In, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.In.In, \frac{1}{2}.In.In.\frac{1}{2},$ $\frac{1}{2}.In.\frac{1}{2}.In.In, \frac{1}{2}.In.\frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.In.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In.In, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.$ $In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In.\frac{1}{2}, \frac{1}{2}.In.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.\frac{1}{2}.In\}$

FIG. 7.12: Les cas de test temporisés générés à partir de la MEFTN de Figure 7.8.

Cependant, avec l'ensemble des hypothèses que nous utilisons (voir section 7.1), nous prouvons que les cas de test générés par notre méthode détecte toutes les fautes introduites dans le chapitre 6. Ceci fait l'object de la prochaine section de ce chapitre.

7.5 Couverture de fautes de la méthode Wp temporisée

Dans cette section, nous étudions la détection de fautes de notre méthode. Pour ce faire, nous passons en revue toutes les fautes introduites dans le chapitre 6 et nous verrons comment les cas de test générés par la méthode Wp temporisée détectent chacune de ces fautes.

7.5.1 Fautes de remise à zéro d'une horloge

La méthode Wp temporisée est basée sur l'architecture de test de la Figure 7.1. Cette architecture rend observable au testeur la remise à zéro d'une horloge. De ce fait, si une implantation ne remet pas à zéro une horloge initialisée à zéro dans la spécification comme le cas de la Figure 6.3, le testeur n'observera pas le signal *ResetClock* attendu à l'interface de la partie contrôle de l'implantation. Ainsi, la faute sera détectée et l'implantation sera déclarée fautive. Similairement, si une implantation remet à zéro une horloge qui doit rester inchangée comme dans la Figure 6.4, le testeur observera le signal *ResetClock* non attendu à l'interface de communication entre la partie contrôle et la partie horloge. Par conséquent, la faute sera détectée et l'implantation sera déclarée fautive.

Proposition 7.6 Détection de fautes de remise à zéro des horloges
Les fautes de remise à zéro des horloges sont détectables par la méthode Wp temporisée grâce à l'architecture de test utilisée.

Preuve
La preuve est simple. Elle est basée sur l'hypothèse que l'implantation utilise le même nombre d'horloges que la spécification (l'hypothèse de test numéro 3).

7.5.2 Fautes de restriction des contraintes temporelles

Comme nous l'avons vu dans la section 6.2.2, la restriction de la contrainte temporelle d'une transition sur une sortie n'est plus considérée comme faute. Cependant, la restriction de la contrainte temporelle d'une transition sur une entrée est une faute. C'est ce type de fautes que nous allons considérer implicitement ou explicitement dans cette section. Notons d'abord que chaque contrainte temporelle d'une transition donne lieu à au moins une région d'horloges (ou un état du système) où la transition est exécutable. Toute restriction de cette contrainte mène alors à une réduction du nombre d'états ou de transitions dans l'implantation. Les cas de test sont générés à partir de l'automate de la spécification où tous les états et les transitions du système sont considérés. Ceci dit que ces cas de test couvrent tous les états et toutes les transitions de la spécification. La proposition suivante résume la relation entre les cas de test et les contraintes des transitions.

Proposition 7.7 Espace temporel des transitions

Pour toute transition sur une entrée, il existe au moins un cas de test qui vérifie l'espace temporel de cette transition.

Preuve

Considérons une transition sur une entrée $l \xrightarrow{?a,R,G} l'$ dans l'AEST. Puisque, dans le test, on ne traite que les spécifications correctes, la transition est exécutable. Alors, il existe au moins un état (l, Z) (Z est une région d'horloges) où la transition est tirable (c'est-à-dire, $\forall v \in Z, v \models G$). Quand on échantillonne le graphe des régions, on obtient au moins un représentant de (l, Z) satisfaisant G. Ceci dit qu'on génère au moins un cas de test vérifiant l'espace temporel de la transition $l \xrightarrow{?a,R,G} l'$.□

À la lumière de cette proposition, si une implantation restreint la contrainte temporelle d'une transition sur une entrée, certains états et/ou transitions seront omis dans son graphe de régions. Ceci est vrai grâce à l'hypothèse numéro 6. Les états et/ou les transitions omis sont couverts par certains cas de test générés par l'algorithme. Alors, lorsqu'on soumet ces cas de test à l'implantation, cette dernière les rejettera

(c'est-à-dire, elle restera dans son état courant) et le testeur n'observera pas la sortie attendue. Par conséquent, on conclut qu'une faute de restriction de la contrainte a été introduite et l'implantation sera déclarée fautive.

Proposition 7.8 Détection de fautes de restriction des contraintes des entrées
Les cas de test générés par la méthode Wp temporisée détectent toutes les fautes de restriction des contraintes temporelles des entrées.

Preuve
Considérons une transition sur une entrée $l_s \xrightarrow{?a,R,G} l'_s$ dans la spécification et supposons que la transition correspondante dans l'implantation est $l_i \xrightarrow{?a,R,G'} l'_i$ tel que G' est une restriction de G. Supposons aussi que la restriction concerne la contrainte sur l'horloge x tel qu'on a $x < m$ dans G et $x < m'$ dans G' $(m' < m)$. Cette modification de la borne supérieure de la contrainte sur x réduit le nombre des régions d'horloges correspondant à G (notées $Regions(G)$). En d'autres termes, l'ensemble $Regions(G) \backslash Regions(G')$ est non vide $(Regions(G) \backslash Regions(G') \neq \emptyset)$. En utilisant les hypothèses 4 et 6, et la proposition 7.7, on conclut que les régions $Regions(G) \backslash Regions(G')$ sont couvertes par au moins un cas de test. Ces derniers seront rejetés par l'implantation. Par conséquent, la faute de restriction de la contrainte G sera détectée et l'implantation sera déclarée fautive. Les cas des formules $x \leq m$, $x = m$, $x > m$ et $x \geq m$ sont prouvés d'une manière similaire. □

7.5.3 Fautes d'élargissement des contraintes temporelles

L'AEST utilisé comme base de spécification des systèmes à étudier a la propriété que toutes les valeurs non pertinentes d'une horloge (c'est-à-dire, les valeurs qui sont supérieures à la plus grande constante utilisée dans les contraintes sur cette horloge) sont représentées par la constante ∞. Lorsqu'on échantillonne le graphe des régions, on obtient au moins un état dans l'automate de grille, et par la suite dans la MEFTN, dans lequel la valeur de l'horloge est ∞. C'est cet état qui sert à vérifier si l'implan-

tation élargit la contrainte temporelle ou pas, particulièrement les contraintes du genre $x < m$. La proposition suivante résume la relation entre l'élargissement de la contrainte d'une transition et l'automate de grille.

Proposition 7.9 Élargissement des contraintes des transitions
Pour chaque transition, $l \xrightarrow{\{?,!\}a,R,G} l'$, *dans l'AEST, l'automate de grille contient au moins un état* (l, v) *tel que* $v \not\models G$.

Preuve
Considérons une transition $l \xrightarrow{\{?,!\}a,R,G} l'$ dans la spécification. La contrainte G donne lieu à au moins une région d'horloges (ou un état du système) où la transition est tirable. Aussi, il existe au moins une région d'horloges (ou un état du système) qui ne satisfait pas la contrainte G. En particulier, si la contrainte G contient une formule $x < m$ (respectivement $x \leq m$), on aura au moins un état dans l'automate de grille où $x = m$ (respectivement $x > m$). Quant aux formules $x \geq m$, $x > m$ et $x = m$, on aura, lorsqu'il est possible, au moins un état dans l'automate de grille dans lequel la valeur de l'horloge x est inférieure à m. En effet, l'horloge prend la valeur $m - \epsilon$, $0 < \epsilon < 1$, avant qu'elle atteint m. □

Puisque l'implantation est supposée ne jamais rejeter des entrées (voir l'hypothèse 3), on peut facilement remarquer que la méthode Wp temporisée teste l'élargissement des contraintes temporelles des transitions. Pour les sorties, l'élargissement de leurs contraintes est détecté en attendant la sortie prévue dans l'intervalle de temps permis. Si le testeur n'observe pas la sortie , il conclut que l'implantation élargit la contrainte temporelle de cette sortie ou elle ne répond pas du tout avec cette sortie. En ce qui concerne les entrées, le testeur les applique dans les états où leurs contraintes temporelles ne sont pas satisfaites afin de vérifier si elles y sont acceptées. Si c'est le cas, l'implantation sera jugée fautive.

Proposition 7.10 Fautes d'élargissement des contraintes
Les cas de test générés par la méthode Wp temporisée détectent toutes les fautes

d'élargissement des contraintes, susceptibles d'exister dans une implantation.

Preuve

Considérons une transition, $l \xrightarrow{?a,R,G} l'$, dans l'AEST. La MEFTN contient au moins un état (l, v) tel que $v \not\models G$. Puisque l'implantation est supposée être complètement spécifiée, chaque état (l, v) a une transition de délai et une transition pour toute entrée de l'AEST. De plus, les hypothèses 3, 4 et 6 garantissent que le nombre d'états dans l'IST est au plus égal à celui de la spécification. Puisque la méthode Wp temporisée couvre tous les états de la MEFTN, on obtient au moins un cas de test pour vérifier l'élargissement de la contrainte de la transition $l \xrightarrow{?a,R,G} l'$. □

7.5.4 Fautes de sorties

Ces fautes sont les plus faciles à détecter. Pour détecter ces fautes, il suffit de traverser toutes les transitions de la MEFTN.

Proposition 7.11 Fautes de sorties

La méthode Wp temporisée détecte les fautes de sorties dans une implantation.

Preuve

La méthode Wp temporisée couvre tous les états et toutes les transitions de la MEFTN. Ceci lui permet de détecter les fautes de sorties. □

7.5.5 Fautes de transfert

Les fautes de transfert sont détectables par toute méthode basée sur l'identification des états [Cho78, FBK+91, SD88b]. Il suffit d'atteindre un état, d'appliquer l'entrée, d'observer la sortie attendue et d'appliquer ensuite la séquence d'identification de l'état d'arrivée de la transition pour vérifier s'il est le bon.

Tout comme les méthodes W [Cho78] et Wp [FBK+91], la méthode Wp temporisée est basée sur la technique de caractérisation des états. Ce qui lui permet de vérifier l'état d'arrivée de chaque transition du système et de détecter ainsi les fautes de transfert.

Proposition 7.12 Fautes de transfert

Les cas de test générés par la méthode Wp temporisée détectent toutes les fautes de transfert dans une implantation.

Preuve

Considérons une transition $t : l_s \xrightarrow{\{?,!\}a,R,G} l'_s$ dans la spécification. Supposons que la transition lui correspondant dans l'implantation est $t' : l_i \xrightarrow{\{?,!\}a,R,G} l'_i$ tel que t' contient une faute de transfert. Grâce aux hypothèses 3, 4 et 6, le nombre d'états dans l'IST est au plus égal à celui de la spécification. Puisque les horloges à remettre à zéro par les deux transitions sont les mêmes (c'est-à-dire, R), l'état d'arrivée (l'_i, v) de t' doit être équivalent à un certain état (l, v) dans la spécification. Supposons que l'état attendu est (l''_i, v). (l''_i, v) est équivalent à l'état (l'_s, v) dans la spécification et dont l'ensemble d'identification est W_j. Par construction, les ensembles W et W_j distinguent les états (l'_s, v) et (l, v). Alors, il existe au moins un cas de test dans $Q.W$ ou $(P \backslash Q).\{W_i\}$ qui détecte la faute de transfert dans t'.□

7.6 Conclusion et discussion

Dans ce chapitre, nous avons introduit la méthode Wp temporisée pour générer des cas de test à partir d'un AEST et nous avons évalué sa couverture de fautes par rapport au modèle de fautes du chapitre 6. La méthode Wp temporisée se base sur la caractérisation des états et utilise la bisimulation temporelle modulo k comme relation de conformité. Pour l'appliquer, nous avons présenté au début de ce chapitre l'ensemble des hypothèses de test que l'IST doit satisfaire et l'architecture de test à utiliser. Cette architecture, quoi qu'elle soit particulière, permet de faciliter le test de remise à zéro des horloges en rendant cette opération observable. Sans elle, certaines

fautes peuvent échapper au test.

Notre approche de génération de tests consiste en plusieurs étapes. Dans la première étape, nous échantillonnons le graphe de régions de la spécification en utilisant une granularité qui ne dépend que du nombre d'horloges. Le résultat de cet échantillonnage est un quotient du graphe des régions que l'on appelle automate de grille. Ce dernier représente explicitement l'écoulement du temps et est suffisant pour tester l'IST par rapport à une bisimulation temporelle modulo k. Dans la deuxième étape, nous transformons l'automate de grille en une machine à états finis temporisée non déterministe observable (MEFTN). Ceci nous permet de réutiliser et d'adapter les techniques de génération de tests basées sur les MEFs. Dans l'étape suivante, la MEFTN est minimisée afin de garantir l'existence de l'ensemble de caractérisation qui permet de distinguer les états entre eux. La minimisation utilisée est légèrement différente de la minimisation classique des MEFs du fait qu'elle doit prendre en considération la sémantique du temps. Dans la dernière étape, nous utilisons la MEFTN résultante de l'étape précédente pour lui appliquer une version adaptée de la méthode Wp généralisée.

Notre méthodologie a été implantée dans un prototype et expérimentée sur des exemples de différentes tailles [Bel98]. La méthode a réussi à générer des cas de test pour les exemples dont la taille est raisonnable. Pour les autres exemples, l'échec de la méthode est principalement dû à la complexité des algorithmes utilisés surtout l'algorithme d'échantillonnage. Ce dernier, rappelons-le, est exponentiel en nombre d'horloges et constantes utilisées dans les contraintes sur ces horloges. Ceci ne limite pas l'applicabilité de notre méthode car les systèmes temps réel existants dont nous sommes au courant utilisent au maximum trois horloges avec des petites constantes entières dans leurs contraintes temporelles.

Comparée aux méthodes [MMM95, Liu93, CL97], notre méthode génère des cas de test à partir d'un modèle riche et général (l'AEST) et garantit une couverture

complète de toutes les fautes étudiées au chapitre 6. Pour ce qui est de la comparaison avec la méthode [SVD01], notre méthode ressemble à cette dernière en ce qui concerne la démarche générale mais elle diffère d'elle de point de vue technique. En ce qui concerne la similarité des deux méthodes, notre méthode et celle de [SVD01] génèrent toutes les deux les cas de test à partir d'un AEST en procédant par une série de transformations et en appliquant des méthodes de caractérisation des états. Les transformations effectuées consistent à réduire le test d'un graphe de régions à un sous-ensemble de ce dernier (l'automate de grille) en utilisant une granularité pour instancier les transitions de délai. Quant à la différence entre les deux méthodes, elle peut être discutée selon les points suivants :

- L'AEST utilisé pour la génération de cas de test. Dans [SVD01], les auteurs utilisent un AEST dans lequel les sorties ne peuvent avoir lieu qu'à des instants entiers. Ceci permet de contrôler les sorties et d'éviter le non-déterminisme dans l'automate de grille. Quant à la méthode Wp temporisée, elle n'impose pas de telles contraintes. Les sorties peuvent avoir lieu à tout moment. Néanmoins, la couverture de tout l'espace temporel des sorties par la granularité d'échantillonnage exige que les contraintes des sorties soient de la forme $x \leq m$, $x = m$, $x > m$ ou $x \geq m$ (voir section 7.4.2).
- La relation de conformité utilisée. Dans [SVD01], les auteurs adoptent la bisimulation comme relation de conformité alors que, dans notre méthode, nous utilisons la bisimulation temporelle modulo k. Le paramètre k correspond à la granularité d'échantillonnage utilisée.
- La granularité utilisée pour construire l'automate de grille. Les auteurs de [SVD01] utilisent une très petite granularité calculée en fonction du nombre de régions d'horloges de la composition de l'automate de la spécification et celui de l'implantation. Cette granularité est 2^{-n},où n est au moins égal au nombre de régions d'horloges de la composition. Ceci suppose que le testeur ait connaissance du nombre d'horloges et les bornes des contraintes utilisés dans l'implantation. On en déduit alors qu'il ne s'agit pas vraiment d'un test boîte

noire. De plus, les auteurs utilisent une borne supérieure de régions d'horloges donnée par Alur et Dill [AD94] pour instancier le paramètre n dans la granularité qu'ils utilisent. Or, cette borne est très grande que le nombre exact calculé dans [EEN98]. Quant à la méthode Wp temporisée, elle utilise au plus une granularité $\frac{1}{(p+2)}$ où p est le nombre d'horloges dans l'automate de la spécification. En comparant cette granularité avec celle utilisée dans [SVD01], il est clair que les tailles des automates de grille obtenus par les deux méthodes sont très différentes.

- La construction de l'automate de grille. Dans notre approche, nous avons donné un algorithme explicite pour construire l'automate de grille alors qu'aucun algorithme n'est donné dans [SVD01].
- La technique utilisée pour générer les cas de test. Dans [SVD01], les auteurs appliquent la méthode W [Cho78] pour générer les cas de test à partir de l'automate de grille alors que, dans notre méthode, nous utilisons une version adaptée de la méthode Wp généralisée [LBP94]. Comme la méthode Wp est une optimisation de la méthode W et vu la différence des tailles des automates de grille utilisés, le nombre de cas de test générés par notre méthode est beaucoup plus petit que celui obtenu par [SVD01].

Dans le chapitre suivant, nous verrons comment la méthode Wp temporisée peut être utilisée pour générer des cas de test pour un système temps réel spécifié par un ensemble d'AESTs communicants.

Chapitre 8

Test des systèmes temps réel encapsulés

Ce chapitre[1] est consacré à la génération des cas de test pour des systèmes temps réel encapsulés. Un système temps réel encapsulé est un ensemble de composantes qui s'exécutent d'une manière concurrente et qui communiquent entre elles sous des contraintes temporelles.

Pour tester un système encapsulé, nous modélisons le système en question par un ensemble d'automates à entrées sorties temporisées communicants. L'un de ces automates spécifie la composante à tester et les autres représentent le contexte de test. La relation entre la composante à tester et les autres composantes doit être prise en considération lors de la génération des cas de test.

Dans ce qui suit, nous discutons d'abord la différence entre le test en contexte et le test en isolation. Ensuite, nous présentons le modèle d'automates à entrées sorties temporisées communicants. Après quoi, nous révisons le modèle de fautes temporelles dans le contexte d'automates à entrées sorties temporisées communicants. Enfin, nous présentons une approche pour tester un automate à entrées sorties temporisées com-

[1] Les résultats de ce chapitre ont été publiés dans [ENKD00].

municant encapsulé en utilisant la méthode Wp temporisée.

8.1 Test en contexte versus test en isolation

Durant la dernière décennie, plusieurs chercheurs se sont intéressés au test des systèmes temps réel afin de généraliser les méthodes de génération de tests basées sur les MEFs et les MEFEs. Les travaux réalisés sont tous fondés sur des modèles temporels isolés tel que les automates temporisés, le graphe des contraintes, les machines à états finis enrichies par des temporisateurs et les logiques temporelles simples. Ces modèles ne permettent pas de décrire tous les systèmes temps réel surtout ceux constitués de plusieurs composantes s'exécutant concurremment et communiquant entre elles par des messages. Ces systèmes ont besoin de modèles plus riches, tel que les automates à entrées sorties temporisées communicants (AESTCs), pour décrire toutes leurs propriétés.

Le test des systèmes temps réel encapsulés modélisés par des AESTCs ne peut se faire par une application directe des méthodes de génération de tests existantes car celles-ci ne permettent pas de tester tous les aspects de ces systèmes. Le développement d'autres techniques devient une chose incontournable. À l'état actuel de nos connaissances, certains travaux ont été réalisés pour tester des MEFs et des MEFEs communicantes [PYBD97, BDAR98] mais il n'existe aucune méthode pour tester des AESTCs.

Dans ce chapitre, nous ne cherchons pas à tester tout le système mais nous nous intéressons au test d'une seule composante du système en présence du reste des composantes. Nous référons à ce type de test par le *test en contexte* ou le *test encapsulé* [ISO91b, PYBD97, BDAR98, LC97].

La Figure 8.1 montre l'architecture des systèmes que nous considérons dans notre étude. Cette architecture est composée de deux entités : la composante à tester notée par l'IST, et le reste de composantes qui forme ce qu'on appelle le *contexte de test.*

Le test en contexte diffère du test en isolation par au moins les points suivants. Premièrement, l'IST n'est pas testée en se basant uniquement sur la spécification ; on doit aussi prendre en considération l'effet du contexte. En effet, le contexte peut tolérer quelques fautes et par conséquent certaines implantations fautives en isolation deviennent conformes à la spécification en contexte. Deuxièment, le contexte n'a pas besoin d'être testé car les composantes du contexte sont supposées être déjà testées en isolation auparavant. Troisiemment, l'exécutabilité de chaque transition dans la spécification n'est pas garantie en présence du contexte de test. Une transition peut ne pas s'exécuter si aucune composante du système ne produit l'action de la transition dans l'intervalle de temps permis. Enfin, puisque l'IST communique avec son contexte via des actions internes et avec son environnement via des actions externes, son comportement en termes de séquences d'entrées/sorties n'est ni complètement contrôlable, ni complètement observable.

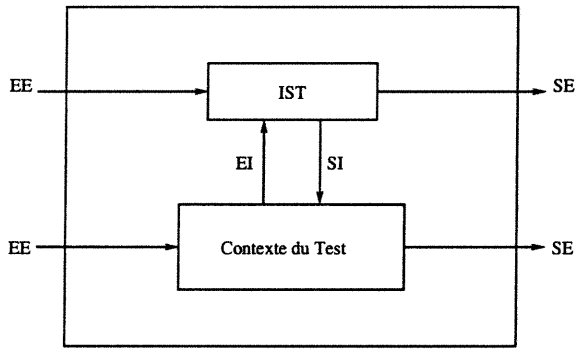

FIG. 8.1: Un système encapsulé.

8.2 Automates à entrées sorties temporisées communicants

Pour décrire formellement les systèmes temps réel encapsulés, nous utilisons les automates à entrées sorties temporisées communicants (AESTCs) dont la définition est comme suit.

Définition 8.1 *Automates à entrées sorties temporisées communicants*
Un automate à entrées sorties temporisées communicant TC est formellement défini par un tuple $TC = (I_{tc}, O_{tc}, L_{tc}, l^0_{tc}, C_{tc}, T_{tc}, F_{tc})$, *où :*

- I_{tc} *est un ensemble fini d'entrées dont chacune commence par "?". On a deux types d'entrées :*
  - *L'ensemble des entrées internes* E_I.
  - *L'ensemble des entrées externes* E_E.
- O_{tc} *est un ensemble fini de sorties dont chacune commence par "!". On a deux types de sorties :*
  - *L'ensemble des sorties internes* S_I.
  - *L'ensemble des sorties externes* S_E.
- L_{tc} *est un ensemble fini d'emplacements.*
- $l^0_{tc} \in L_{ct}$ *est l'emplacement initial.*
- C_{tc} *est un ensemble fini d'horloges, initialisées toutes à zéro en* l^0_{ct}.
- T_{tc} *est un ensemble fini de transitions.*
- F_{tc} *est un canal FIFO.*

Les transitions d'un AESTC sont similaires à celles d'un AEST. Elles sont formées de la même manière et respectent les mêmes contraintes. De même, le domaine temporel de chaque horloge $x \in C_{tc}$ est un intervalle $[0, C_x] \cup \{\infty\}$, où C_x est la plus grande constante entière utilisée dans les contraintes sur l'horloge x.

Un système temps réel est défini par une collection d'AESTCs $(TC_1, TC_2, ..., TC_n)$, où chaque TC_i, $1 \leq i \leq n$, représente le comportement d'un processus du système. La communication entre les différents processus du système est assujettie aux hypothèses suivantes.

- Une action ne peut être à la fois interne et externe. Nous avons alors $E_E \cap S_E = \emptyset$ et $E_I \cap S_I = \emptyset$.
- Un AESTC ne peut jamais communiquer avec lui-même ($\forall i = 1, 2, ..., n$ $E_{TC_i} \cap S_{TC_i} = \emptyset$).
- Chaque action interne n'a qu'un seul émetteur et qu'un seul récepteur.
- Pour chaque entrée interne $?a$ (respectivement une sortie interne $!a$) dans l'AESTC TC_i, il existe une sortie interne $!a$ (respectivement une entrée interne $?a$) qui lui correspond dans un autre AESTC TC_j $(i \neq j)$.
- La communication entre les différents processus du système se fait d'une manière asynchrone via des canaux fiables et bornés. Sans perte de généralité, nous supposons, dans le reste de ce chapitre, que la longueur de tous les canaux de communication est de 1.
- Toutes les composantes du système commencent en même temps leurs exécutions.

La Figure 8.2 montre trois AESTCs. Ils interagissent avec leur environnement via les entrées externes $?e1$ et $?e2$ et communiquent entre eux via les entrées internes $?i1$ et $?i2$ (les sorties internes $!i1$ et $!i2$). Dans cet exemple, il n'y a aucune sortie externe.

Le comportement de tout le système est décrit par un AEST global représentant les emplacements et les transitions globaux du système. Un emplacement global d'un système temps réel $(TC_1, TC_2, ..., CT_n)$ est un tuple $(l_1, l_2, ..., l_n, F_1, F_2, ..., F_n)$, où l_i, $1 \leq i \leq n$, est un emplacement dans TC_i et F_i, $1 \leq i \leq n$, est la file FIFO associée à TC_i. Une transition globale d'un système temps réel $(TC_1, TC_2, ..., CT_n)$ est une transition entre deux emplacements globaux causée par l'exécution d'une transition dans une composante du système. L'AEST global, T_g, d'un système temps réel $(TC_1, TC_2, ..., CT_n)$ est obtenu en calculant le produit cartésien de tous les AESTCs

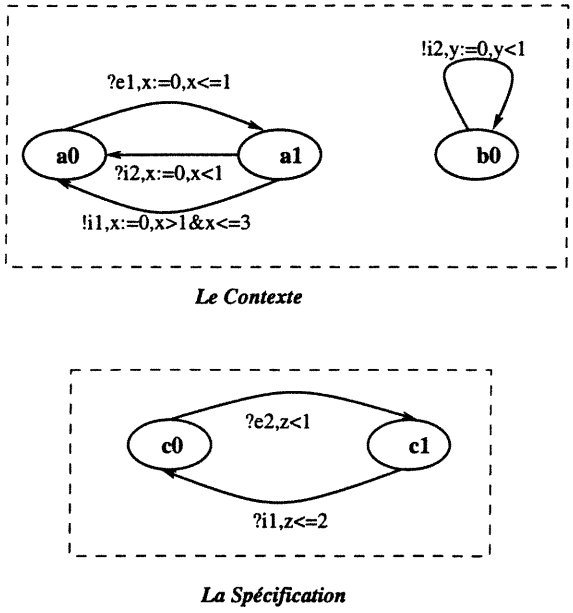

FIG. 8.2: Un exemple d'AESTCs.

décrivant le système ($T_g = TC_1 \times TC_2 \times ... \times TC_n$). La sémantique d'une collection d'AESTCs ($TC_1, TC_2, ..., CT_n$) est donnée par le graphe de régions de son AEST global.

La construction du produit cartésien de tous les AESTCs est coûteuse car elle est exponentielle en nombres d'emplacements et de transitions des AESTCs. De plus, nous avons souvent besoin de ne construire qu'un produit partiel du système à partir d'un petit ensemble de transitions $TS = \{t_1, t_2, ..., t_n\}$.

Définition 8.2 *Produit partiel des AESTCs*
Soit $(TC_1, TC_2, ..., TC_n)$ *une collection d'AESTCs décrivant un système temps réel. Le produit partiel d'un AESTC* $TC_i, 1 \leq i \leq n$, *noté* $PP(TC_i)_{1 \leq i \leq n}$, *sur l'ensemble de transitions* $TS = \{t_1, t_2, ..., t_n\}$ *est défini par* $TC_i \times TS$. *Notons que* $PP(TC_i) \subseteq (TC_1 \times TC_2 \times ... \times TC_n)$.

8.3 Révision du modèle de fautes temporelles

Comme nous l'avons mentionné dans les chapitres 2 et 6, le modèle de fautes est très dépendant du modèle de spécification utilisé pour la génération des cas de test. Dans cette section, nous révisons le modèle de fautes temporelles dans le contexte d'AESTC. Nous illustrons surtout l'effet du contexte sur les fautes dans une composante donnée et la propagation des fautes entre les différentes composantes du système.

Le modèle de fautes temporelles consiste, rappelons le, en deux types de fautes : les fautes temporelles et les fautes de sorties et de transfert. Les fautes temporelles affectent les contraintes temporelles des transitions. Pour la famille des automates temporisés, une faute temporelle peut être soit une remise à zéro d'une horloge, une non remise à zéro d'une horloge, une restriction ou un élargissement de la contrainte d'une transition. Une faute de sorties affecte la sortie de la transition. Une faute de transfert, quant à elle, change l'état d'arrivée d'une transition. Dans le cas d'un AESTC, nous nous intéressons uniquement à la restriction et à l'élargissement des contraintes sur les entrées et les sorties internes. Les autres types de fautes ne nécessitent pas de révision car ils sont soit observables par l'environnement soit non influencés par le contexte de test.

8.3.1 Fautes temporelles sur les entrées internes

À cause du contexte, la restriction et l'élargissement des contraintes temporelles ne sont pas toujours considérées comme fautes. En ce qui concerne la restriction, le contexte de test peut produire une sortie interne uniquement dans un sous-espace temporel de l'entrée interne correspondante dans la spécification. Toute restriction de la contrainte temporelle de cette entrée interne à ce sous-espace temporel n'est pas une faute. Considérons la transition $c_1 \xrightarrow{?i_1, z \leq 2} c_0$ de la spécification de la Figure 8.2. Voici deux types de restriction de la contrainte temporelle de cette transition :

- $c_1 \xrightarrow{?i_1, z \leq 1} c_0$. Cette restriction est une faute effective car les états $(c_1, 1 < z \leq 2)$

sont tous accessibles. La séquence $\frac{1}{4}.?e_1.\frac{1}{4}.?e_2.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}$ est une trace qui amène le système de l'état $(a_0, c_0, x = z = 0)$ à l'état $(a_0, c_0, x = \frac{5}{4} \& z = \frac{6}{4})$.

- $c_1 \xrightarrow{?i_1, z>1 \& z<2} c_0$. Cette restriction n'est pas une faute effective car la transition $c_1 \xrightarrow{?i_1, 1>z \& z<2} c_0$ est équivalente en contexte à la transition $c_1 \xrightarrow{?i_1, z \leq 2} c_0$. En d'autres termes, Il n'existe aucune trace qui peut amener le système de son état initial à un état où $z \leq 1$.

D'une manière similaire, même si une implantation élargit la contrainte temporelle d'une transition sur une entrée interne, l'espace temporel additionnel ne peut être accessible si le contexte produit toujours la sortie interne dans un sous-espace de l'espace temporel initial. Considérons de nouveau la transition $c_1 \xrightarrow{?i_1, z \leq 2} c_0$ de la spécification de la Figure 8.2. L'élargissement de sa contrainte temporelle par $c_1 \xrightarrow{?i_1, z>2 \& z \leq 4} c_0$ est une faute effective car l'état $(2 < z < 3)$ est accessible à partir de l'état initial du système. Si on change, maintenant, la transition $c_1 \xrightarrow{?i_1, z \leq 2} c_0$ de la spécification de la Figure 8.2 par la transition $c_1 \xrightarrow{?i_1, z \leq 4} c_0$, la faute $c_1 \xrightarrow{?i_1, z \leq 4} c_0$ n'est pas effective car l'état $(c_1, z = 4)$ n'est pas accessible en contexte. Ceci dit qu'il n'existe aucune trace qui peut amener le système de son état initial $(a_0, c_0, x = z = 0)$ à un état $(a_i, c_1, x = v, z = 4)$. On dit alors que la transition $c_1 \xrightarrow{?i_1, z \leq 4} c_0$ est équivalente en contexte à la transition $c_1 \xrightarrow{?i_1, z \leq 4} c_0$.

8.3.2 Fautes temporelles sur les sorties internes

Tout comme les entrées, la restriction et l'élargissement des contraintes temporelles des transitions sur les sorties ne sont pas toujours considérées comme fautes, en présence d'un contexte. En effet, une implantation qui restreint la contrainte temporelle d'une transition sur une sortie n'est plus considérée fautive si le contexte n'accepte pas la sortie dans le sous-espace temporel manquant.

D'une manière similaire, on ne parle plus de faute si l'implantation élargit la contrainte temporelle d'une transition sur une sortie et que le contexte n'accepte pas la sortie dans le sous-espace temporel additionnel. Pour clarifier ces propos, recon-

sidérons la Figure 8.2 et remplaçons respectivement les transitions $c_1 \xrightarrow{?i_1,z\leq 2} c_0$ et $a_1 \xrightarrow{!i_1,x:=0,x>1\&x\leq 3} a_0$ par $c_1 \xrightarrow{!i_1,z>2\&z\leq 4} c_0$ et $a_1 \xrightarrow{?i_1,x:=0,x>1\&x\leq 3} a_0$. Si une implantation remplace la transition $c_1 \xrightarrow{!i_1,z>2\&z\leq 4} c_0$ par $c_1 \xrightarrow{!i_1,z>2\&z\leq 3} c_0$ et que le contexte n'accepte pas l'entrée interne $?i_1$ quand $3 < z \leq 4$, l'implantation en question sera déclarée conforme à sa spécification. De même, si une implantation élargit la contrainte temporelle de la transition $c_1 \xrightarrow{!i_1,z>2\&z\leq 4} c_0$ par $c_1 \xrightarrow{!i_1,z>2\&z\leq 5} c_0$ et que le contexte n'accepte pas l'entrée interne $?i_1$ quand $4 < z \leq 5$, l'implantation en question sera déclarée conforme à sa spécification.

8.4 Génération de tests à partir d'un AESTC encapsulé

Une façon simple de tester un AESTC encapsulé consiste à composer, d'abord, tous les AESTCs du système en un seul et à générer ensuite les cas de test à partir de l'automate global résultat. Cette solution assure que toutes les fonctionnalités du système soient testées et garantit ainsi une couverture complète des fautes. Cependant, la construction de l'automate global du système donne lieu au problème d'explosion d'états (ou explosion combinatoire). De plus, le contexte de test n'a pas besoin d'être entièrement testé. Pour cela, nous faisons un compromis entre la couverture des fautes et le coût de la composition de tout le système. Nous visons ainsi atteindre une bonne couverture de fautes en ne construisant qu'un produit partiel du système.

Notre approche, comme la montre la Figure 8.3, est formée de trois étapes principales :

- La sélection des transitions du contexte qui affectent (ou qui sont affectées par) l'exécution de la composante à tester (la spécification).
- La construction du produit partiel du système.
- L'application de la méthode Wp temporisée sur le produit partiel résultat.

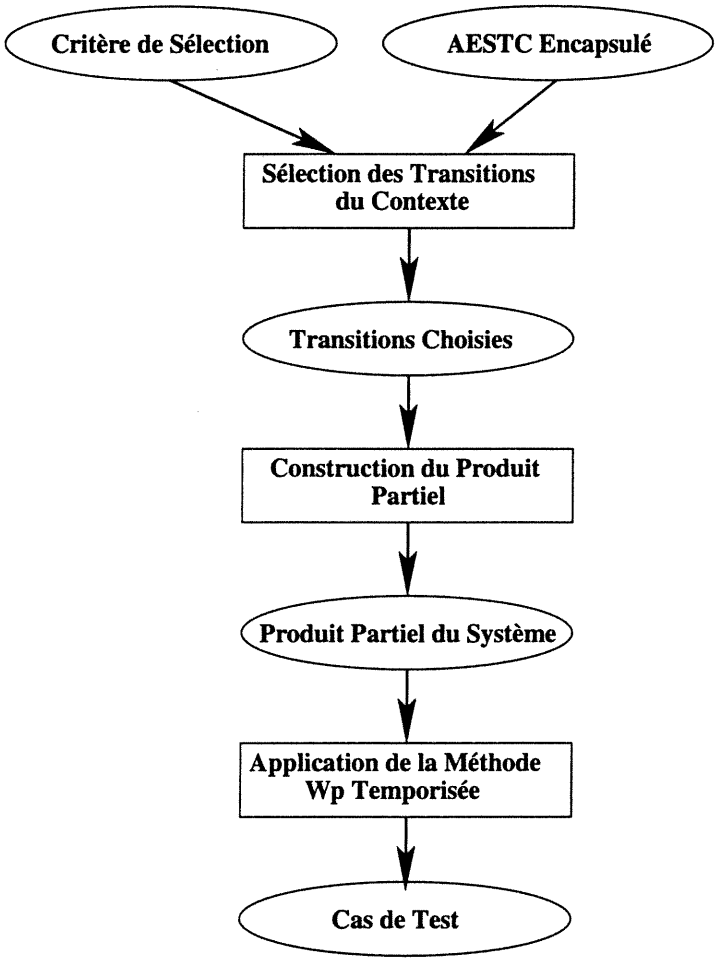

FIG. 8.3: Un aperçu général du test d'un AESTC encapsulé.

8.4.1 Sélection des transitions du contexte

Cette étape consiste à chercher les transitions de la spécification et celles du contexte à considérer pour tester une implantation en contexte. L'approche est récursive et se base sur la technique de marquage [BDAR98].

Puisque nous voulons tester une implantation par rapport à sa spécification, toutes les transitions de la spécification sont d'abord marquées. Pour chaque transition marquée t_s, nous examinons l'ensemble des chemins du contexte qui peuvent induire l'exécution de la transition t_s. Chaque chemin est une séquence de transitions consécutives ; il est formé de trois parties : la transition t_c correspondant à la transition t_s, le préambule qui amène le système de l'emplacement initial à l'emplacement de départ de t_c, et le postambule qui commence à l'emplacement d'arrivée de t_c et se termine à un certain emplacement l_i. Toutes les transitions des chemins sélectionnés seront marquées. La procédure est répétée récursivement jusqu'à ce que toutes les transitions marquées soient traitées.

Lors de la recherche d'un chemin correspondant à une transition marquée, nous essayons de finir le chemin par une transition sur une sortie externe afin de pouvoir observer les fautes possibles dans les transitions internes. Si de telles sorties n'existent pas, certaines fautes risqueront de rester non détectées dans l'implantation.

Le choix d'un chemin ou d'un autre est très important car il affecte la qualité de test de l'implantation. En fait, pour chaque transition marquée t_s, il se peut qu'il existe une infinité de chemins pouvant induire l'exécution de t_s. Ceci est principalement dû à l'infinité des préambules se terminant à l'emplacement de départ de la transition t_c, correspondante à t_s. Le choix de l'un de ces préambules dépend fortement de l'objectif de test. Pour étudier cet aspect, nous nous basons sur l'exécutabilité des chemins et leurs capacités de détection de fautes, pour définir ci-après quelques critères de choix de préambules à utiliser.

Considérons une transition t_s dans la spécification et la transition correspondante t_c dans le contexte. Nous choisissons uniquement les préambules permettant à la transition t_c de s'exécuter. Les autres préambules mènent à un blocage du système dans l'emplacement de départ de la transition t_c. Cette condition est appelée *critère d'exécutabilité.* L'ensemble des préambules exécutables est obtenu en propageant la contrainte temporelle de chaque transition vers son emplacement d'arrivée [Som97] et en vérifiant la satisfaction de la contrainte temporelle de toute transition sortante de cet emplacement. Ce processus est répété jusqu'à ce que la transition t_c soit atteinte.

En plus de ce critère, nous définissons deux autres critères simples. Le *critère de plus court préambule* consiste à choisir le plus court préambule exécutable permettant d'atteindre l'emplacement source de la transition t_c. Le *critère du meilleur préambule*, quant à lui, consiste à choisir le meilleur préambule exécutable permettant de tester la plus grande partie de l'espace temporel de la transition t_c. Il est clair que ce critère est plus fort que le critère de plus court préambule mais il est plus complexe que ce dernier.

Dans le reste de ce chapitre, nous utilisons le critère de plus court préambule pour choisir les chemins du contexte qui affectent (ou qui sont affectés par) une transition t_s dans la spécification.

En suivant l'approche proposée, nous utilisons uniquement les transitions du contexte qui sont pertinentes à la génération des cas de test. Nous évitons ainsi le problème d'explosion d'états dû à la construction du produit cartésien de la spécification et du contexte. Notons que le test d'un système modélisé par deux AESTCs (un représentant la spécification et l'autre le contexte) est équivalent au test du produit cartésien des deux automates car toutes les transitions du contexte vont être marquées et sélectionnées par l'approche. Ceci dit que la méthode proposée ne devient intéressante que lorsque le nombre d'AESTCs est supérieur à deux.

La Figure 8.4 décrit l'algorithme de sélection des transitions du contexte à considérer pour la génération des cas de test. Notre algorithme prend comme entrées un critère de sélection de transitions et un ensemble d'AESTCs. L'un de ces automates décrit la composante à tester alors que le reste définit le contexte de test. Comme sorties, l'algorithme fournit un ensemble de transitions du contexte qui satisfont le critère de sélection et qui seront utilisées pour la génération des cas de test. Dans le pire cas, l'ensemble des transitions sélectionnées sera formé de toutes les transitions du contexte. Cependant, dans la plus part des cas, cet ensemble est beaucoup plus petit.

Début
MarkedTransitions = Toutes les transitions de la spécification.
HandledTransitions = $\emptyset$.
Tant que $MarkedTransitions \backslash HandledTransitions \neq \emptyset$ **Faire**
Choisir une transition t_s de $MarkedTransitions \backslash HandledTransitions$.
Ajouter t_s à $HandledTransitions$.
Soit t_c la transition du contexte correspondante à t_s.
Soient $Source$ et $Target$ respectivement les emplacements
de départ et d'arrivée de la transition t_c.
Trouver un préambule dans le contexte qui se termine à $Source$
et satisfait le critère de sélection.
Trouver un postambule de la transition t_c contenant au moins
une sortie externe.
Marquer la transition t_c et les transitions du préambule et
du postambule tout en les ajoutant a $MarkedTransitions$
FinTantQue
FinAlgorithme

FIG. 8.4: L'algorithme de sélection des transitions du contexte.

8.4.2 Construction du produit partiel

Après application de l'étape précédente, nous obtenons un ensemble de transitions qui affectent (ou qui sont affectées par) l'exécution de la composante à tester. Ces transitions sont utilisées ensuite pour calculer un produit partiel du système conformément à la définition 8.2. Le produit partiel qu'on obtient est généralement

beaucoup plus petit que le produit cartésien du système. L'algorithme de construction du produit partiel d'un système par rapport à un ensemble de transitions $\{t_1, t_2, ..., t_n\}$ est donné à la Figure 8.5.

Début
Soit $(CT)_{1 \leq i \leq n}$, un ensemble d'AESTCs.
LocationsSet = $(l_1^0, l_2^0, ..., l_n^0, \varepsilon, \varepsilon, ..., \varepsilon)$ (l'emplacement initial du système).
TransitionsSet = $\emptyset$.
HandledLocations = $\emptyset$.
Tant que $LocationsSet \backslash HandledLocations \neq \emptyset$ **Faire**
Choisir un emplacement $l = (l_i^1, l_i^2, ..., l_i^j, ..., l_i^n, F_i^1, F_i^2, ..., F_i^j, ..., F_i^n)$
de $LocationsSet \backslash HandledLocations$.
S'il existe une transition marquée $l_i^j \xrightarrow{\{?,!\}a,R,G} l_k^j$ Alors
Si a est une entrée interne et $F_i^j = a.F_k^j$
Alors $F_i^j = F_k^j$
Sinon si a est une sortie interne et le canal F_j n'est
pas plein Alors $F_i^j = F_i^j.a$
FinSi
Créer un nouvel emplacement $l = (l_i^1, l_i^2, ..., l_k^j, ..., l_i^n, F_i^1, F_i^2, ..., F_i^j, ..., F_i^n)$
et l'ajouter à $LocationsSet$ s'il n'existe pas
Ajouter la transition correspondante $(l_i^1, l_i^2, ..., l_i^j, ..., l_i^n, F_i^1, F_i^2, ..., F_i^j, ..., F_i^n)$
$\xrightarrow{\{?,!\}a,R,G}$ $(l_i^1, l_i^2, ..., l_k^j, ..., l_i^n, F_i^1, F_i^2, ..., F_i^j, ..., F_i^n)$
à $TransitionsSet$.
FinTantQue
FinAlgorithme

FIG. 8.5: L'algorithme de construction d'un produit partiel.

La taille du produit partiel résultat dépend principalement du nombre de transitions sélectionnées dans la section précédente. Dans le pire cas, le produit partiel est égal au produit cartésien du système tout entier. Cependant, dans la plus part des cas, seulement une petite portion de ce dernier est construite. D'un autre côté, la qualité du produit partiel est mesurée par la couverture de fautes des cas de test générés à partir de lui. Cette couverture de fautes est fondamentalement liée au critère de sélection utilisé pour choisir les transitions du contexte à considérer dans la construction du produit partiel du système. Plus fort est le critère de sélection, meilleure est la qualité du produit partiel du système.

La Figure 8.6 montre le produit partiel résultant de l'application des deux étapes précédentes sur l'exemple de la Figure 8.2.

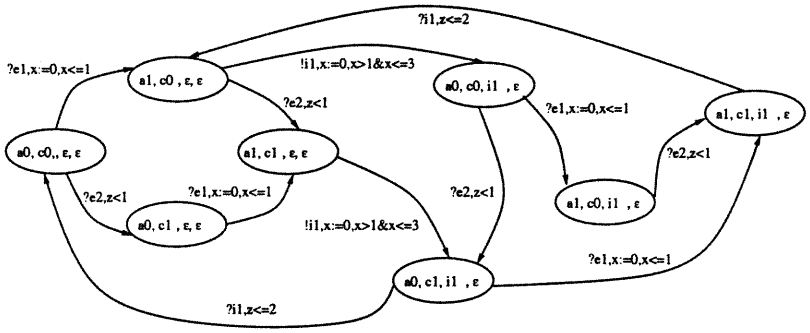

FIG. 8.6: Le produit partiel des AESTCs de la Figure 8.2.

8.4.3 Application de la méthode Wp temporisée

Le produit partiel obtenu dans l'étape précédente est un automate à entrées sorties temporisées sur lequel on peut appliquer une méthode de génération de tests temporisés. Dans notre approche, nous choisissons la méthode Wp temporisée, décrite dans le chapitre précédent, pour générer des cas de test à partir du produit partiel résultat. Comme nous l'avons vu dans le chapitre précédent, cette génération de cas de test temporisés se fait en quatre étapes.

- L'échantillonnage du graphe de régions correspondant au produit partiel. Nous utilisons une granularité qui ne dépend que du nombre d'horloges dans l'automate du produit partiel. Dans l'exemple de la Figure 8.6, nous utilisons une granularité de $\frac{1}{4}$.
- La transformation de l'automate de grille résultant de l'échantillonnage en une machine à états finis temporisée non-déterministe (MEFTN).
- La minimisation de la MEFTN tel qu'elle est décrite dans le chapitre précédent.
- L'application de la méthode Wp temporisée.

La couverture de fautes de notre approche dépend de la qualité du produit partiel construit. Évidemment, les cas de test générés ne couvrent pas tout le modèle de fautes discuté dans la section 8.3 mais ils détectent toutes les fautes liées au produit partiel. La non-couverture de certaines fautes est principalement due au critère utilisé pour sélectionner les transitions du contexte à considérer dans la construction du produit partiel du système.

8.5 Conclusion

Nous avons présenté dans ce chapitre une méthode pour tester une composante temps réel encapsulée. Les systèmes temps réel que nous avons considérés sont modélisés par un ensemble d'automates à entrées sorties temporisées communicants (AESTCs). L'un de ces automates spécifie la composante à tester et le reste représente son contexte. Nous avons introduit au début le modèle des AESTCs et nous avons vu la composition complète et partielle d'un ensemble d'AESTCs. Ensuite, nous avons révisé le modèle de fautes temporelles dans le contexte d'AESTCs. Finalement, nous avons proposé une approche pour tester un AESTC encapsulé.

Notre approche est basée sur la construction d'un produit partiel de la spécification et de son contexte. Pour cela, nous sélectionnons, selon le critère de plus court préambule, certains (et non pas tous les) chemins du contexte qui affectent (ou qui sont affectés par) les transitions de la spécification. Les cas de test sont générés à partir du produit partiel résultat en utilisant la méthode Wp temporisée.

La méthode que nous avons proposée génère des cas de test temporisés exécutables avec une bonne couverture de fautes. Cette couverture peut être améliorée davantage en choisissant un critère fort pour sélectionner les transitions à considérer dans la construction du produit partiel. Par exemple, si on choisit le critère du meilleur préambule, on aura sans aucun doute une meilleure couverture de fautes. Le problème est que ce critère est plus complexe et risque de mener à une explosion d'états. Il y a

alors un compromis à faire entre la complexité du critère adopté et la couverture de fautes des cas de test générés.

Chapitre 9

Conclusion

Dans cette thèse, nous avons développé une méthodologie pour le test des systèmes temps réel modélisés par des automates à entrées sorties temporisées qu'ils soient communicants ou pas. Pour ce faire, nous avons commencé par introduire le sujet et présenter les différents éléments nécessaires pour notre recherche. Comme nous nous intéressons au test de conformité, nous avons présenté dans le chapitre 2 les différentes étapes à suivre pour tester la conformité d'une implantation par rapport à sa spécification de référence. Nous avons expliqué la méthodologie OSI et nous avons introduit les concepts nécessaires à son automatisation et qui sont utilisés dans la pratique du test de conformité. Nous avons surtout parlé des modèles de spécification, des relations de conformité, des modèles de fautes, des hypothèses de test et de la couverture de tests. Comme la spécification formelle est le point de départ de toute automatisation du test de conformité, nous sommes revenus sur ce point dans le chapitre 3 pour présenter les principaux modèles formels utilisés pour spécifier les systèmes informatiques. Selon notre sujet de recherche, nous avons utilisé l'expressivité temporelle comme critère pour classer les modèles formels en deux catégories : les modèles non-temporels et les modèles temporels. La première classe regroupe les modèles dont l'expressivité temporelle est restreinte (MEF, MEFE, STE, etc...) alors que la deuxième classe inclut les modèles décrivant suffisamment le comportement temporel des systèmes. Dans le chapitre 4, nous avons présenté la génération de cas

de test à partir des modèles en relation avec notre sujet de recherche. Ainsi, nous avons présenté les méthodes de génération de tests basées sur les MEFs. Certaines de ces méthodes sont utilisées dans les derniers chapitres de cette thèse. Ensuite, nous avons présenté le test basé sur les MEFEs. Le point commun entre le test des MEFEs et le test des AESTs est l'existence de certaines variables dans les transitions qui rendent certains cas de test non-exécutables. Enfin, dans le même chapitre, nous avons passé en revue les méthodes de génération de tests temporisés. Effectivement, ces méthodes génèrent des cas de test pour les systèmes temps réel mais elles souffrent de certains problèmes. Elles sont soit restrictives utilisant des modèles simples pour spécifier les systèmes temps réel, soit elles ne détectent pas toutes les erreurs possibles dans l'implantation ou soit elles génèrent un nombre exorbitant des cas test même pour un système simple. Dans le chapitre 5, nous avons introduit le modèle des automates à entrées sorties temporisées que nous utilisons pour modéliser les systèmes temps réel qu'on veut tester et nous avons discuté les problèmes posés lors de la génération des cas de test pour ces systèmes. Dans les chapitres 6, 7 et 8, nous avons présenté nos principales contributions. Ci-après, nous résumons ces résultats et nous présentons les extensions possibles à notre recherche.

9.1 Contributions principales

Au cours de ce travail, nous avons proposé de nouvelles approches pour tester les systèmes temps réel modélisés par des automates à entrées sorties temporisées qu'ils soient communicants ou pas. Nos principales contributions sont : le modèle de fautes temporelles, la génération de cas de test à partir d'un automate à entrées sorties temporisées et la génération de cas de test à partir d'un automate à entrées sorties temporisées communicant.

9.1.1 Modèle de fautes temporelles

Le modèle de fautes joue un rôle important dans le test des systèmes informatiques que ce soient matériels ou logiciels. Il regroupe l'ensemble des fautes simples susceptibles d'exister dans une implantation d'un système. Le modèle de fautes fournit non seulement une base pour la définition des mesures de couverture de test mais il sert aussi au développement des méthodes de génération de tests et l'analyse des résultats de test pour des fins de diagnostique. Quoique les principes de modèles de fautes soient essentiellement les mêmes pour le matériel, les logiciels et les protocoles de communication, les modèles de fautes à appliquer dépendent fortement du formalisme de spécification décrivant l'implantation sous test. Dans le chapitre 6, nous avons proposé un modèle de fautes pour les systèmes temps réel modélisés par des automates à entrées sorties temporisées. Dans ce modèle, nous ne nous sommes intéressés qu'aux fautes simples et nous les avons classées en deux catégories : les fautes temporelles qui affectent les contraintes temporelles des transitions (remise à zéro d'une horloge, non remise à zéro d'une horloge, restriction des contraintes et élargissement des contraintes) et les fautes de sorties et de transfert qui affectent respectivement les sorties et les emplacements d'arrivée des transitions. Nous avons aussi distingué les fautes effectives des fautes non-effectives.

9.1.2 Génération de cas de test à partir d'un AEST

La génération de tests joue un rôle très important dans le processus du test de conformité (voir chapitre 2). Elle permet de dériver, à partir d'une spécification, les cas de test à soumettre à l'implantation qu'on veut tester afin de juger sa conformité par rapport à sa spécification de référence. Dans le chapitre 7, nous avons étudié le test temporisé et nous avons développé une approche permettant de générer des cas de test temporisés à partir d'un AEST. Cette approche consiste en plusieurs étapes. Dans la première étape, nous échantillonnons le graphe de régions de la spécification en utilisant une granularité qui ne dépend que du nombre d'horloges. Le résultat de cet échantillonnage est un quotient du graphe de régions que l'on appelle au-

tomate de grille. Ce dernier représente explicitement l'écoulement du temps et est suffisant pour tester l'IST par rapport à une bisimulation temporelle modulo k. Dans la deuxième étape, nous transformons l'automate de grille en une machine à états finis temporisée non déterministe observable (MEFTN). Ceci nous permet de réutiliser et d'adapter les techniques de génération de tests basées sur les MEFs. Dans l'étape suivante, la MEFTN est minimisée afin de garantir l'existence de l'ensemble de caractérisation qui permet de distinguer les états entre eux. La minimisation utilisée est légèrement différente de la minimisation classique des MEFs dans le sens où on prend en considération la sémantique du temps. Dans la dernière étape, nous appliquons la méthode Wp temporisée à la MEFTN minimale résultante de l'étape précédente. La méthode Wp temporisée est une version adaptée de la méthode Wp généralisée où le temps est pris en compte. La méthode Wp temporisée a une couverture complète de fautes recensées dans le modèle de fautes présenté dans le chapitre 6. Ceci est en partie garanti par l'utilisation de l'architecture de test que nous avons développée et présentée dans le chapitre 7. Cette architecture ressemble à un test boîte grise et consiste à séparer le comportement du système à étudier en deux parties : la partie contrôle qui prend en considération la communication avec l'environnement en termes de messages d'entrées/sorties et la partie horloge qui modélise l'aspect temporel du système. La communication entre les deux parties se fait via des signaux internes pour consulter et remettre à zéro les valeurs des horloges. Dans cette architecture, le seul signal interne observable par le testeur est le signal de remise à zéro des horloges.

9.1.3 Génération de cas de test à partir d'un AESTC

La plupart des systèmes temps réel sont composés de plusieurs processus s'exécutant concurremment et communicant les uns avec les autres via des canaux FIFO sous des contraintes temporelles. Dans le chapitre 8, nous avons étudié le test de ce type de systèmes et nous l'avons comparé au test en isolation. Pour tester ce type de systèmes, nous avons choisi de les modéliser d'abord par des AESTCs et de tester ensuite une composante en présence des autres composantes. Ce type de test, connu sous le nom

du test en contexte ou test encapsulé, diffère du test en isolation et nécessite de nouvelles méthodes de génération de tests. Notre approche est basée sur la construction d'un produit partiel de la spécification et de son contexte. Nous sélectionnons, selon un certain critère, certains (et non pas tous les) chemins du contexte qui affectent (ou qui sont affectés par) les transitions de la spécification. Les cas de test sont générés à partir du produit partiel résultat en utilisant la méthode Wp temporisée. La méthode que nous avons proposée génère des cas de test temporisés exécutables avec une bonne couverture de fautes.

Dans le même chapitre, nous avons également révisé le modèle de fautes temporelles dans le contexte d'un AESTC. Nous avons surtout illustré l'effet du contexte sur le test de l'IST et la propagation des fautes entre les différentes composantes du système à tester.

9.2 Extensions possibles

Le travail que nous avons réalisé dans cette thèse apporte des solutions au test des systèmes temps réel modélisés par des automates à entrées sorties temporisées (communicants). Cependant, notre travail peut faire l'objet de plusieurs extensions. Ces extensions permettraient d'appliquer nos méthodologies à une large gamme des systèmes temps réel et d'étudier plus en détail la couverture de fautes de notre approche de test en contexte. Plus précisément, il serait intéressant de :

- Adapter et étendre nos approches afin de générer des cas de test à partir d'un automate hybride contenant en même temps des variables d'horloges et des variables de données.
- Étudier en détail la couverture de fautes de notre approche de génération de tests pour les systèmes temps réel encapsulés. Il s'agit de définir des mesures quantitatives et qualitatives de la couverture de fautes des cas de test générés à partir d'un AESTC encapsulé.

- Utiliser les objectifs de test pour ne tester que les parties critiques d'un système temps réel modélisé par un AEST et réduire ainsi le nombre de cas de test générés par notre approche.
- Détailler la relation entre le critère de sélection de transitions du contexte et la couverture de fautes des cas de test générés à partir d'un AESTC.
- Attaquer la décomposition et la distribution des cas de test, générés à partir d'un AESTC, entre les différentes composantes du système et de résoudre le problème de synchronisation de ces composantes lors de l'exécution du test.

Bibliographie

[ACH94] R. Alur, C. Courcoubetis, and T.A. Henzinger. The Observational Power of Clocks. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 1994.

[AD94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[Bô2] R. Bûchi. On a Decision Method in Restricted Second-Order Arithmetic. In *International Congress on Logic, Methodology, and Phylosophy of Science*, pages 1–12. Stanford University Press, 1962.

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1) :25–29, January 1988.

[BD87] T. Budkowski and P. Dembinski. An Introduction to Estelle : A Specification Language for Distributed Systems. *Computer networks ans ISDN systems*, 14(1), 1987.

[BDA96] C. Bourhfir, R. Dssouli, and E. Aboulhamid. Automatic Test Generation for EFSM-based Systems. Technical report TR1043, Université de Montréal, August 1996.

[BDAR97] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic Executable Test Case Generation for EFSM Specified Protocols. In Chapman and Hall, editors, *Proceedings of the 10th International Workshop of Communicating Systems IWTCS'98 (Cheju Island, Korea)*, 1997.

[BDAR98] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. A Guided Incremental Test Case Generation for Conformance Testing for CEFSM Specified Protocols. In Kluwer Academic Publishers, editor, *Proceedings of the 11th International Workshop of Communicating Systems IWTCS'98 (Tomsk. Russia)*, September 1998.

[BDD+91] G. V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault Models in Testing. In J. Kroon, R. J. Heijink, and E. Brinksma, editors, *Proceedings of the International Workshop on Protocol Test System IFIP*. North-Holland, October 1991.

[BDD+92] G. V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault Model in Testing. In *Proceedings of the 4th IFIP TC6 International Workshop Protocol Test System*, 1992.

[Bel89] F. Belina. The CCITT-Specification and Description Language SDL. *Computer networks ans ISDN systems*, 16(4) :331–341, 1989.

[Bel98] F. Belqasmi. *Génération de Tests pour les Systèmes Temps-Réel.* Rapport de Stage, ENSIAS, Université Mohammed V et DIRO, Université de Montréal, June 1998.

[Bou99] C. Bourhfir. *Génération Automatique de Cas de Tests pour les Sysèmes Modélisés par des Machines à Etats Finis Communicantes.* Phd thesis, Université de Montréal, April 1999.

[Bri88] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Proceedings of the 8th International Symposium on Protocol Specification, Testing and Verification.* IFIP, North-Holland, 1988.

[BSS86] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specification, Their Implementations and Their Tests. In B. Sarikaya and G.v. Bochmann, editors, *Protocol Specification, Testing, and Verification, VI,* Montreal, Québec, Canada, June 10-13, 1986, pages 349–360. North-Holland, 1986.

[CA97] Ana R. Cavalli and R. Anido. Verification and Testing Techniques Based on Finite State Machine Model. Research Report 97-09-02, INT, France, 1997.

[CCI93] CCITT. CCITT, Specification and Description Language (SDL), Recommendation Z.100. International Standard Z.100, CCITT, Genève, 1993.

[Čer92a] K. Čerāns. *Algorithmic Problems in Analysis of Real Time System Specifications.* Dr.sc.comp. thesis, University of Latvia, Rīga, 1992.

[Čer92b] K. Čerāns. Decidability of Bisimulation Equivalences for Parallel Timer Processes. In G. von Bochmann and D.K. Probst, editors, *Proceedings of the 4th International Workshop on Computer Aided Verification,* Montreal, Canada, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1992.

[Cha97] O. Charles. *Application des Hypothèsde Test à une Définition de la Couverture.* PhD thesis, Université Henri Poincaré-Nancy 1, 1997.

[Cho74] Y. Choueka. Theories of Automata on ω-Tapes : A Simplified Approach. *Computer System Science*, 8, 1974.

[Cho78] T. S. Chow. Testing Software Design Modeled by Finite State Machine. *IEEE Transactions Software Engineering*, SE-4, n3 :178–187, 1978.

[CL97] D. Clarke and I. Lee. Automatic Generation of Tests for Timing Constraints from Requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems,* Newport Beach, California, February 1997.

[Cla96] D. Clarke. *Testing Real-Time Constraints.* PhD Thesis, University of Pennsylvania, 1996.

[CMS94] A. Cerone and A. Maggiolo-Schettini. Expressivity of Timed Petri Nets : Classical versus Time-Critical Viewpoint. In *The Workshop on Concurrency, Specification and Programming*, pages 63–84, Poland, 1994. Warsaw Univ. Publishing Co., Warsaw.

[CR83] J. E. Coholahan and N. Roussopoulos. Timed Requirements for Timed-Driven Systems using Augmented Petri Nets. *IEEE transactions on Communication Protocols*, 9 :603–616, 1983.

[CZ93] S. T. Chanson and J. Zhu. A Unified Approach to Protocol Test Sequence Generation. In *IEEE INFOCOM, San Fransisco, USA*, 1993.

[Das85] B. Dasarathy. Timing Constraints of Real-Time Systems : Constructs for Expressing Them, Methods of Validating Them. *IEEE transactions on Software Engineering*, 11(1) :80–86, January 1985.

[DCDMPS82] F. De Cindio, G. De Michelis, L. Pomello, and C. Simone. Superposed Automata Nets. In C. Girault and W. Reisig, editors, *Application and Theory of Petri Nets*, number 52 in Informatik Fachberichte, pages 229–279. Springer-Verlag, 1982.

[Dil90] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems,* Grenoble, France, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1990.

[DSA+99] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir. Test Development for Communication Protocols : Towards Automation. *A Special Issue of Computer Networks ans ISDN Systems*, 31 :1835–1872, 1999.

[DY95] C. Daws and S. Yovine. Two Examples of Verification of Multirate Timed Automata with KRONOS. In *Proceedings of the 1995 IEEE Real-Time Systems Symposium, RTSS'95,* Pisa, Italy. IEEE Computer Society Press, 1995.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 1996 IEEE Real-Time Systems Symposium, RTSS'96,* Washington DC, USA. IEEE Computer Society Press, 1996.

[EEN98] A. Elqortobi and A. En-Nouaary. Dénombrement du Nombre des Régions dans un Automate Temporisé. Technical Report TR-1116, Département IRO, Universite de Montréal, Montréal, Canada, January 1998.

[EJ73] J. Edmonds and E. L. Johnson. Matching Euler Tours and the Chinese Postman. *Mathematical Programming*, 5 :88–124, 1973.

[ENDE97] A. En-Nouaary, R. Dssouli, and A. Elqortobi. Génération de Tests Temporisés. In *Proceedings of the* 6^{th} *Colloque Francophone de l'ingénierie des Protocoles, HERMES, ISBN 2-86601-639-4*, 1997.

[ENDK01] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-Method : Testing Real-Time Systems. *Under Correction in IEEE Transactions on Software Engineering*, April 2001.

[ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterisation Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain*, December, 2-4 1998.

[ENKD99a] A. En-Nouaary, F. Khendek, and R. Dssouli. Fault Coverage in Testing Real-Time Systems. In *6th International Conference on Real-Time Systems Computing Systems and Applications (RTCSA'99), Hong Kong*, December, 13-15 1999.

[ENKD99b] A. En-Nouaary, F. Khendek, and R. Dssouli. Fault Coverage in Testing Real-time Systems. technical report TR-1162, Département IRO, Université de Montréal, September 1999.

[ENKD00] A. En-Nouaary, F. Khendek, and R. Dssouli. Testing Embedded Real-Time Systems. In *7th International Conference on Real-Time Systems Computing Systems and Applications (RTCSA'00), Cheju Island, South Korea*, December, 12-14 2000.

[FBK+91] S. Fujiwara, G.V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite-State Models. *IEEE Transactions Software Engineering*, SE-17, NO. 6 :591–603, 1991.

[Gil61] A. Gill. State-Identification Experiments in Finite Automata. *Information and Control*, 4 :132–154, 1961.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE transactions on Communication Protocols*, 17 :160–172, 1991.

[Gon70] G. Gonenc. A Method for the Design of Fault Detection Experiment. *IEEE transactions on Computers*, C-19 :551–558, 1970.

[HLJ95] C. M. Huang, Y. C. Lin, and M. Y. Jang. Executable Data Flow and Control Flow Protocol Test Sequence Generation for EFSM-Specified Protocol. In *International Workshop On Protocol Test Systems (IWPTS'95), Evry, France*, 1995.

[ISO86] ISO. *Estelle : a formal description technique based on an extended state transition model* ISO/TC97/SC21/WG16–1 DP9074, 1986. ISO/TC97/SC21/WG16–1 DP9074.

[ISO88] ISO. LOTOS — A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO91a] ISO. Conformance Testing Methodology and Framework. International Standard IS-9646 9646, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1991.

[ISO91b] ISO9646. Information processing systems - open systems interconnection - osi conformance testing methodology and framework, part 1 : General concepts, part 2 : Abstract test specification, part 3 : The

tree and tabular combined notation (ttcn), part 4 : Test realization, part 5 : Requirements on test laboratories and clients for the conformance assessment process. International Standard 9646, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1991.

[KAD+00] A. Khoumsi, M. Akalay, R. Dssouli, A. En-Nouaary, and L. Granger. An Approach For Testing Real-Time Protocols. In *TESTCOM Ottawa, Canada*, August-September 2000.

[Kan95] I. Kang. *CTSM A Formalism for Real-Time System Analysis based on State-Space Exploration.* PhD Thesis, University of Pennsylvania, 1995.

[KC95] M. Kim and S. T. Chanson. Design for Testability of Protocols Based on Formal Specifications. In *International Workshop On Protocol Test Systems (IWPTS'95), Evry, France*, 1995.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Co., 1978.

[LBP94] G. Luo, G. V. Bochmann, and A. Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method. *IEEE Transactions Software Engineering*, SE-20, NO. 2 :149–162, 1994.

[LC97] L. P. Lima and A. Cavalli. A Pragmatic Approach to Generating Test Sequences for Embedded Systems. In *Proceedings of the International Workshop on Testing Communica ting Systems (IWTCS'97), Cheju Islands, Korea*, 1997.

[LDB+93] G. Luo, R. Dssouli, G. V. Bochmann, P. Venkataram, and A. Ghedamsi. Generating synchronizable Test Sequences based on State Machine with Distributed Ports. In *Proceedings of the 6th IFIP TC6 International Workshop Protocol Test System*, September 1993.

[Liu93] F. Liu. Test Generation based on an FSM Model with Timers and Counters. Master thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1993.

[LY93] K.G. Larsen and W. Yi. Time Abstracted Bisimulation : Implicit Specification and Decidability. In *Proceedings Mathematical Foundations of Programming Semantics (MFPS 9)*, volume 802 of *Lecture Notes in Computer Science*, New Orleans, USA, April 1993. Springer-Verlag.

[Mil80a] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mil80b] Robin Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, vol. 92, 1980.

[Mil89a] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mil89b] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4) :365–398, November 1995.

[Nah95] R. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts.* Phd thesis, University of Bern, March 1995.

[NSY92] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Compiling Real-Time Specifications into Extended Automata. *IEEE transactions on Software Engineering*, 18(9) :794–804, September 1992.

[NT81] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition- Tours. *Proceedings of Fault Tolerant Computer Systems*, pages 238–243, 1981.

[Pet91] A. Petrenko. Checking Experiments with Protocol Machines. In J. Kroon, R. J. Heijink, and E. Brinksma, editors, *Proceedings of the*

International Workshop on Protocol Test System IFIP. North-Holland, October 1991.

[PYBD97] A. Petrenko, N. Yevtushenko, G. V. Bochmann, and R. Dssouli. Testing in Context : Framework and Test Derivation. *A Special Issue on Protocol Engineering of Computer Communication*, 1997.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems Using Petri Nets*. Phd thesis, project mac, mac-tr 120, MIT, 1974.

[Ray87] D. Rayner. OSI Conformance Testing. *Computer networks ans ISDN systems*, 14 :79–98, 1987.

[RDT95a] T. Ramalingom, A. Das, and K. Thulasiraman. A Unified Test Case Generation Method for the EFSM Model using Context Independent Unique Sequences. In *International Workshop On Protocol Test Systems (IWPTS), Evry, France*, 1995.

[RDT95b] T. Ramalingom, A. Das, and K. Thulasiraman. A unified test case generation method for the efsm model using context independent unique sequences. In *Proceeding Of the International Workshop on Protocol Test System (IWPTS'95), Evry, France*, 1995.

[Rei85] W. Reisig. *Petri Nets : An Introduction*. EATCS Monographs on Theoretical Computer Science, Volume 4. Springer-Verlag, 1985.

[Rei87] W. Reisig. Petri Nets in Software Engineering. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets : Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course,* Bad Honnef, September 1986, volume 255 of *Lecture Notes in Computer Science*, pages 63–96. Springer-Verlag, 1987.

[RT86] G. Rozenberg and P.S. Thiagarajan. Petri Nets : Basic Notions, Structure, Behaviour. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 585–668. Springer-Verlag, 1986.

[SD88a] K. Sabnani and A. Dabhura. A Protocol Test Generation Procedure. *Computer networks ans ISDN systems*, 15 :285–297, 1988.

[SD88b] K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. *Computer Networks and ISDN Systems*, 15 :285–297, 1988.

[Som97] S. T. Somé. *Un Cadre d'Ingénierie des Exigences par Scénarios d'Interaction.* Phd thesis, chapitre 6, Université de Montréal, October 1997.

[SVD01] J. Springintveld, F. Vaadranger, and P. Dargenio. Testing Timed Automata. *Theoretical Computer Science*, 254 :225–257, 2001.

[Tay80] B. Taylor. Introducing Real-time Constraints into Requirements and High Level Design of Operating Systems. In *In Proceedings 1980 Nat. Telecommunications Conference,Houston, TX, volume 1*, pages 18.5.1–18.5.5, 1980.

[Tre92a] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, Department of Computer Science, University of Twente, December 1992.

[Tre92b] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD Thesis, University of Twente, August 1992.

[UY91] H. Ural and B. Yang. A test sequence selection method for protocol testing. *IEEE Transactions on Communication,*, 39(4), 1991.

[VCI89] S. T. Vuong, W. W. L. Chan, and M. R. Ito. The UIOv Method for Protocol Test Sequence Generation. In *Proceedings of the 2-nd International Workshop Protocol Test System, Berlin, Germany*, 1989.

[Wal83] B. Walter. Timed Petri Net for Modelling and Analyzing Protocols with Real-Time Characteristics. In H. Rudin and C. H. West, editors, *The 3rd IFIP Workshop on Protocol Specification, Testing and Verification*, pages 149–159, Amsterdam, 1983. North-Holland.

[WR85] E. J. Weyuker and S. Rapps. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering,*, SE-11(4), 1985.

[Yao95] M. Yao. *On the Development of Conformance Test Suites in View of Their Fault Coverage.* PhD thesis, Université de Montréal, Département IRO, 1995.

[Yi90] W. Yi. Real-Time Behaviour of Asynchronous Agents. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer-Verlag, 1990.