Université de Montréal

Diagnostic des systèmes temps réel modélisés par des automates à entrées sorties temporisées

Présenté par:

Khalid El Ghazouani

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

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maîtrise es science (M.Sc.)
En informatique

Décembre, 2002

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :

Diagnostic des systèmes temps réel modélisés par des automates à entrées sorties temporisées

Présenté par :

Khalid El Ghazouani

A été évalué par un jury composé des personnes suivantes :

Julie Vachon
Président-rapporteur

Rachida Dssouli
Directrice de recherche

Mustapha Nourelfath
Codirecteur de recherche

Michel Gendreau
Membre du jury

Mémoire accepté le 23 janvier 2003

Sommaire

Dans ce mémoire, nous proposons une méthode de diagnostic pour les systèmes temps réel modélisés par des Automates à Entrée Sortie Temporisées (AESTs). Nous traitons le cas où une faute de sortie ou de transfert pourrait être dans une transition de l'implantation. Notre méthode permet de localiser la transition fautive dans l'AEST suite à une détection par la méthode de test Wp temporisée.

Le processus consiste en six étapes principales. Dans la première étape, nous transformons l'AEST en une Machine à États Finis Temporisée Non déterministe (MEFTN) observable et facilement testable. Dans la deuxième étape, nous exécutons chaque cas de test sur l'implantation, puis sur la MEFTN pour générer les sorties attendues. Dans la troisième étape, nous comparons les sorties observées à celles attendues. Si des symptômes sont détectés, nous générons un ensemble minimal de transitions suspectes de la MEFTN qui peuvent expliquer les anomalies dans le comportement de l'implantation sous diagnostic. Par la suite, nous identifions la (ou les) transitions qui correspondent à chacune de ces transitions suspectes dans l'AEST en utilisant deux algorithmes de transformation. Dans la quatrième étape, nous analysons les symptômes relatifs aux différents cas de test pour déduire des hypothèses (qui expliqueraient les erreurs détectées) et générer un ensemble réduit de transitions suspectes. Dans la cinquième étape, pour chacune de ces dernières, nous construisons les AESTs (dits machines mutantes) représentant son comportement erroné. Nous appliquons ensuite les cas de test sur ces automates pour comparer les sorties nouvellement attendues à celles déjà observées de l'implantation. Seules les transitions suspectes pour lesquelles les machines mutantes fournissent les mêmes sorties observées sont des candidats plausibles. Dans la dernière étape, s'il y a plusieurs machines mutantes qui représentent le comportement de l'implantation, nous générons des cas de test additionnels pour discriminer entre les candidats maintenus dans la dernière étape. Nous développons aussi un outil qui implante la méthode proposée.

Mots-Clés : Diagnostic – Test – Automates Temporisés – Systèmes temps réel – Modèle de fautes.

Abstract

In this work, we propose a method of diagnosis for real time systems modeled by Timed Input Output Automata (TIOA). We deal with the case where one transfer or output fault may be present in a transition of the real-time system implementation. Timed Wp-method is used to generate timed test cases.

The process consists in six main steps. In the first step we transform the TIOA into an observable Non-deterministic Finite State Machine (NTFSM) easily testable. In the second step, we apply each test case to the implementation to generate observed outputs, then on the specification to generate expected outputs. In the third step, we compare those expected outputs with the observed ones for each test case. If symptoms are detected, we generate a minimal set of suspect transitions from the NTFSM that can explain the dysfunction in the behavior of the implementation under diagnosis. Thereafter, we identify the corresponding transitions, to each one of these suspect transitions, in the TIOA by using two transformation algorithms. In the fourth step, we analyze the symptoms relating to the various test cases to deduce assumptions (which would explain the detected errors) and to generate a reduced set of suspect transitions. In the fifth step, for each one of those transitions, we build the TIOA (called 'mutant' machines) representing its erroneous behavior. Then, we apply the test cases to these machines to compare the newly expected outputs to the already observed ones from the implementation. Only the suspect transitions for which a mutant machine releases the same observed outputs are candidates. In the last step, if there is several mutant machines which represent the behavior of the implementation, we generate additional test cases to discriminate between the candidates maintained in the last step. We also propose a tool that implements the suggested method.

KEY WORDS : Diagnosis – Test – Specification – Implementation – Timed automata – Real-time systems.

Table des matières

Liste des Figures

Liste des Tableaux

Remerciements

J'exprime ma profonde gratitude au Professeur Rachida Dssouli pour avoir dirigé ce travail avec confiance et enthousiasme et pour ses conseils pertinents. Mes sincères remerciements à mon codirecteur le Professeur Mustapha Nourelfath de l'Université de Québec en Abitibi Témiscamingue pour son assistance et pour son appui considérable tout au long de ce travail.

Je tiens à remercier vivement le Professeur Abdeslam En-Nouaary pour le temps qu'il m'a toujours accordé ainsi que de m'avoir guidé dans mes démarches.

Je remercie les Professeurs : Julie Vachon et Michel Gendreau pour l'honneur qu'ils me font en faisant partie du jury de mon mémoire.

Un grand remerciement à tous les membres du groupe téléinformatique du département d'informatique et de recherche opérationnelle de l'Université de Montréal qui grâce à leur esprit de collaboration ont contribué directement ou indirectement à la réalisation de ce travail.

Chapitre 1

Introduction

Le test joue un rôle très important dans le milieu industriel. En effet, cette activité est associée à tout processus qui génère un produit. C'est un ensemble d'étapes qui déterminent si le produit va être commercialisé ou non. La phase de test permet de garantir la qualité d'un produit avant de le délivrer. Elle consiste à valider le comportement d'une implantation par rapport à un ensemble de données d'entrées, appelées *cas de test*. Ces derniers sont sélectionnés auparavant dans l'objectif de stimuler les différentes fautes et observer leurs manifestations. Il vient par la suite la localisation et la correction des erreurs détectées. Ces deux dernières étapes sont regroupées dans une seule appelée *débogage* ou *diagnostic.*

Le diagnostic a été débattu dans plusieurs domaines tels que les circuits logiques, l'hydromécanique et la médecine. Son processus est composé de trois étapes fondamentales :

- Application des données de test au système sous observation.
- Comparaison des résultats de test avec ceux attendus.
- Analyse et explication des problèmes et des erreurs détectées.

Dans la littérature, plusieurs méthodes ont été proposées pour le diagnostic d'un système. Ces méthodes dépendent du type du système et de la méthode de spécification. On distingue principalement deux classes de diagnostic :

- Diagnostic expérimental utilisé essentiellement en médecine et en d'autres domaines similaires.

- Diagnostic basé sur un modèle ou encore le diagnostic basé sur la structure ou le comportement.

Dans ce deuxième type de diagnostic, il est important de savoir comment le système est supposé se comporter afin de déterminer pourquoi il ne fonctionne pas correctement. Pour décrire le comportement d'un système, plusieurs méthodes de spécification basées sur différents modèles ont été proposées.

Dans le domaine des protocoles de communication, on trouve des méthodes orientées vers les transitions d'états telles que celles qui se basent sur les Machines à États Finis (MEFs) [BOCH 78] et les réseaux de Petri [MERL 79], et celles qui se basent sur les langages formels et la logique temporelle [SCHW 82].

Dans un processus de diagnostic, l'implantation est tout d'abord testée afin de vérifier sa conformité par rapport à la spécification. Cette phase est appelée *test de conformité*. Plusieurs méthodes ont été développées pour la génération des cas de test, permettant ainsi de détecter le maximum de fautes dans une Implantation Sous Test (IST). Les mêmes cas de test peuvent être utilisés pour la localisation des fautes détectées. Une approche utilisée, dans le cadre des MEFs, consiste à analyser les résultats de l'application des cas de test ; puis générer l'ensemble des transitions plausibles pouvant expliquer le comportement observé. Des cas de test supplémentaires peuvent être appliqués à l'IST pour réduire l'espace des 'suspects'. Une deuxième approche permet de localiser les fautes à partir de la construction de la machine qui traduit le comportement observé. Quelques hypothèses peuvent aider à réduire la complexité du processus de diagnostic. Nous pouvons, par exemple, supposer qu'une et une seule faute se présente à la fois, ce qui permet d'optimiser le nombre de chemins parcourus lors de la localisation des fautes.

1.1 Objectif et contribution

Dans le cadre de ce travail, nous proposons de développer une méthode de diagnostic des systèmes temps réel pour localiser certaines fautes détectées dans la phase de test. Les systèmes que nous étudierons sont modélisés par des Automates à Entrées Sorties Temporisées (AESTs) pour décrire la communication entre le système et son

environnement ainsi que les contraintes régissant cette communication. Les AESTs sont une variante des automates d'Alur et Dill [ALUR 94] dans lesquels l'alphabet est subdivisé en un ensemble d'entrées que le système reçoit de son environnement et un ensemble de sorties représentant les messages que le système produit à son environnement.
Vu le rôle primordial que la phase de test joue dans un processus de diagnostic, nous utiliserons la méthode Wp temporisée, récemment proposée [ENNO 01], pour le test de ce type de systèmes. Cette méthode permet une meilleure couverture de fautes.

Le diagnostic des systèmes temps réel diffère de celui des systèmes non temporisés à cause des contraintes temporelles dans leur comportement. De ce fait, les méthodes de diagnostic basées sur les MEFs (étendus ou non) ne sont pas directement applicables pour les AESTs. Il faut alors étendre le diagnostic non temporisé pour prendre en compte la notion et la sémantique du temps.

Notre but est de développer une méthodologie nouvelle et pratique pour le diagnostic des fautes de sortie et de transfert dans une IST spécifiée sous forme d'un AEST. Plus précisément, nous visons à :

- Présenter le diagnostic des systèmes non temporisés.
- Étudier le modèle de fautes des systèmes temps réel modélisés par les AESTs.
- Étudier l'effet des fautes de sortie et de transfert sur le comportement d'une implantation modélisée par un AEST.
- Développer une méthode d'analyse de ce comportement.
- Localiser les fautes de sortie et de transfert détectées par la méthode Wp temporisée.
- Proposer des algorithmes pour automatiser les étapes de la méthode de diagnostic.
- Développer un outil informatique pour le diagnostic de ces fautes.

1.2 Organisation du mémoire

Le reste de ce mémoire est organisé de la manière suivante. Le chapitre 2 introduit le problème de diagnostic d'une manière générale, comme il a été défini au sein de la communauté de l'intelligence artificielle. Quelques méthodes sont présentées en détail dans ce chapitre. Ces dernières sont importantes dans la mesure où elles présentent les tâches indispensables dans n'importe quel processus de diagnostic. Elles permettent aussi d'organiser les différentes étapes requises pour le diagnostic des systèmes présentés par des modèles.

Dans le troisième chapitre, nous présentons un état de l'art sur le diagnostic des protocoles de communication modélisés par les MEFs et les Machines à États Finis Étendues(MEFEs). Nous détaillons en particulier deux méthodes de diagnostic qui correspondent à ces deux types de représentations. Ces méthodes seront la base de notre contribution au diagnostic des systèmes temps réel modélisés par des AESTs.

Au quatrième chapitre, nous passons en revue la liste des méthodes de génération des cas de test temporisé basé sur les modèles formelles et nous discutons les points forts et les lacunes de chaque méthode. Cette discussion nous conduit à choisir la méthode que nous utiliserons dans notre approche de diagnostic des systèmes temps réel. Le choix d'une telle méthode sera fondé sur sa complétude et son degré de couverture des fautes.

Dans le cinquième chapitre, nous détaillons la méthode proposée pour le diagnostic des systèmes temps réel modélisés par les AESTs. Cette méthode permet de localiser une faute de sortie ou de transfert dans ce type de systèmes. Les algorithmes développés et les différentes étapes de notre proposition seront illustrées à travers un exemple.

Le dernier chapitre présente l'outil de diagnostic que nous avons développé pour implémenter la méthode proposée. Nous détaillerons en particulier le modèle conceptuel de cet outil, ses différentes composantes et nous donnerons un exemple d'exécution.

Nous terminerons ce rapport par une conclusion générale.

Chapitre 2

Approches de diagnostic

Le problème de diagnostic est un sujet bien documenté dans le domaine de l'intelligence artificielle. Il a été développé dans plusieurs domaines d'applications tels que les circuits logiques, les systèmes mécaniques complexes et la médecine. Ainsi, différents systèmes de raisonnement basés sur des modèles de spécification ont été développés dans ce sens comme le GDE pour 'General Diagnostic Engine' [KLEE 87] et HT pour 'Hardware Troubleshooting'.

Ce chapitre présente l'état de l'art en matière de diagnostic. Nous commencerons par une description des différents concepts d'un processus de diagnostic. Ensuite, nous présenterons deux approches qui étaient à la base de plusieurs processus de diagnostic dans la littérature, notamment la méthode de Davis [DAVI 82] et l'approche de Reiter [REIT 87]. Enfin, nous détaillerons les étapes fondamentales d'un algorithme général de diagnostic [GHED 93a].

2.1 Présentation générale d'un processus de diagnostic

Un processus de diagnostic contient principalement les trois étapes suivantes :

- Application des mesures ou des cas de test à un système donné.
- Détection des résultats de test qui sont différents de ceux attendus.
- Identification et explication des causes des fautes, en utilisant les mesures et les résultats de test.

D'une manière générale, le processus de diagnostic est une tâche difficile, spécialement pour les systèmes complexes. Pour faciliter le problème, il devient nécessaire d'utiliser un modèle de fautes [BOCH 91]. Étant donné une description hiérarchique d'un système, un modèle de fautes peut être établi en utilisant les différents niveaux d'abstraction. Quelques modèles peuvent déterminer toutes les défaillances de chaque composante du système. Par exemple, dans le cas des systèmes modélisés par des MEFs, un modèle de fautes basé sur les fautes de sortie et les fautes de transfert peut être efficace pour un processus de diagnostic [CHOW 78] [GHED 93b] [VUON 90].

Pour la localisation des fautes, tout processus de diagnostic exécute deux tâches principales : La génération des candidats plausibles et la discrimination entre les candidats.

2.1.1 Génération des candidats plausibles

À partir des symptômes observés et du modèle de spécification, le processus de diagnostic déduit des hypothèses sur les causes éventuelles des fautes. Chaque hypothèse explique une faute dans une ou plusieurs composantes du système. Un bon générateur de candidats doit être complet, non redondant et optimal. On dit qu'il est complet s'il génère tous les candidats susceptibles d'expliquer les observations ; il est non redondant s'il ne génère pas le même candidat plus d'une fois et il est optimal s'il génère un ensemble minimal de candidats et non un sur-ensemble de candidats.

2.1.2 Discrimination entre les candidats

La phase de génération des candidats plausibles produit un très grand nombre de candidats susceptibles d'expliquer les observations. Pour réduire cet espace, deux techniques peuvent être utilisées. La première consiste à générer des cas de test supplémentaires, appelés tests de distinction (ou distinguishing test). La deuxième technique consiste à choisir de nouveaux points d'observations dans le système sous diagnostic afin de le rendre facilement testable et donc diagnosticable.

2.2 La méthode de Davis

Dans [DAVI 82], l'auteur a étudié le problème de diagnostic en traitant le cas d'une faute simple dans un circuit logique. Le processus de diagnostic est basé sur la structure et la description du comportement du circuit. Le comportement de chaque composante est représenté par des contraintes qui décrivent la relation entre les entrées et les sorties du système. Nous distinguons entre deux types de contraintes. Le premier type, appelé *contraintes de simulation*, décrit comment les composantes sont supposées fonctionner. Le deuxième type, appelé *contraintes d'inférence*, est utilisé pour déduire les valeurs des variables afin de vérifier si ces dernières ne sont pas en contradiction avec l'ensemble des contraintes. La table suivante contient les contraintes décrivant le comportement des composantes du circuit de la Figure 2.1

Composantes / *Contraintes*	**Mult-1**	**Mult-2**	**Mult-3**	**Add-1**	**Add-2**
Contraintes de simulation	X= A*C	Y= B*D	Z=C*E	F=X+Y	G=Y+Z
Contraintes d'inférence	A=X/C C=X/A	B=Y/D D=Y/B	C=Z/E E=Z/C	X=F-Y Y=F-X	Y=G-Z Z=G-Y

Tableau 2.1 : L'ensemble des contraintes pour le circuit de la figure 2.1

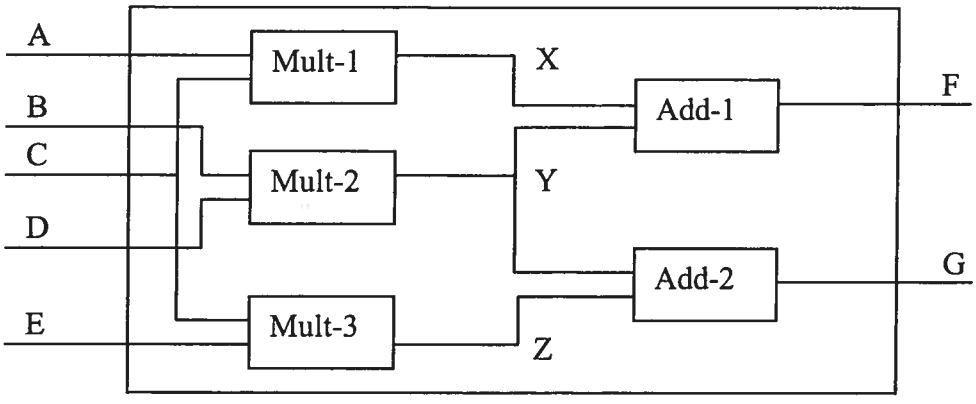

Figure 2.1 : Un circuit avec trois multiplicateurs et deux additionneurs

2.2.1 Processus de diagnostic

Le processus de diagnostic procède en deux phases. La première est la génération des candidats diagnostics. Elle consiste à déterminer les composantes fautives qui peuvent expliquer les sorties observées. La deuxième phase quant à elle permet de réduire l'espace des candidats générés lors de la première phase en appliquant au système les séquences de test de distinction appropriées.

Génération des candidats diagnostics

Cette première phase contient trois étapes :

Étape 1 : Identification des symptômes

La suite de test sélectionnée est appliquée parallèlement au circuit pour observer le comportement du système sous diagnostic et aux contraintes de simulation pour décrire le comportement attendu de ce système. Une comparaison entre les sorties observées et les sorties attendues permet de générer les symptômes.

Étape 2 : Générations des suspects

Pour chaque symptôme on remonte le chemin dans le réseau des contraintes afin d'identifier toutes les composantes pouvant participer à l'apparition du symptôme. Chaque composante dans ce chemin est considérée comme un suspect. Une fois tous les symptômes sont considérés, l'ensemble résultant contient tous les suspects qui pourront expliquer tous les symptômes.

Étape 3 : Sélection des candidats diagnostics

Pour chaque suspect S, toutes les contraintes reliées à S sont retirées du réseau des contraintes décrivant le système. Les sorties observées sont insérées dans le réseau réduit, et les contraintes du nouveau réseau sont ensuite exécutées. Si le réseau est dans un état consistent, alors S est confirmé comme un candidat diagnostic. Par contre, si des contradictions sont encore obtenues, le suspect S sera éliminé de la liste des candidats.

Discrimination entre les candidats

Sous l'hypothèse d'une faute simple, seulement un candidat, parmi ceux qui sont générés à la première phase, représente la composante fautive.

Pour isoler ce candidat, de nouveaux tests doivent être sélectionnés et appliqués au système. Dans [DAVI 88], l'auteur propose une approche dans ce sens.

2.3 La théorie de Reiter

Dans [REIT 87], l'auteur a développé une théorie générale de diagnostic basé sur un modèle. Il a proposé un algorithme permettant de déterminer tous les diagnostics qui peuvent expliquer les conflits entre le comportement prédit et un comportement observé. Son algorithme traite aussi bien le cas d'une faute simple que des fautes multiples. Le processus de diagnostic est basé sur la structure et le comportement du système décrit par un modèle. Ce processus utilise les notions d'ensemble de conflits et d'ensemble de candidats.

Nous commencerons par une synthèse de la théorie de Reiter [REIT 87] en introduisant quelques définitions et théorèmes, puis nous présenterons l'algorithme ainsi qu'un exemple d'application.

2.3.1 Définitions

Le système sous diagnostic est représenté par une paire (SD, Composants) telle que :

- **SD** est la description du système en logique du premier ordre.
- Le terme **Composants** désigne les composants du système exprimés par un ensemble fini de constantes.

Une observation d'un système est un ensemble fini d'expressions du premier ordre. On écrit (SD, Composants, OBS) pour un système (SD, Composants) sous une observation OBS.

Un ensemble H est dit un 'Hitting Set' pour un ensemble C de collection d'ensembles si $H \subseteq \cup \{s \in C \mid H \cap s \neq \varnothing$ pour chaque $s \in C\}$.

H est dit minimal si $\forall H' \subset H$, H' n'est pas un 'Hitting Set'.

Le diagnostic d'un système (SD, Composants, OBS) est défini comme étant l'ensemble minimal $\Delta \subseteq$ Composants tel que :

SD ∪ OBS ∪ { ¬ AB(c) | c ∈ Composants - Δ } ∪ { AB(c) | c ∈ Δ } est consistent. AB(c) est prédicat indiquant que le composant c se comporte anormalement. En d'autre terme, il existe une explication logique pour les observations OBS, quand les composants fonctionnent normalement, à l'exception de ceux de Δ.

La méthode de diagnostic est basée sur la détermination des ensembles 'Hitting Set' minimaux pour les ensembles de conflits définis comme suit :
Si le système ne fonctionne pas correctement, les observations seront en conflit avec ce que le modèle du système prédit. Or, Le modèle prédit que tous les composants fonctionnent correctement, ce qui peut être exprimé comme suit :
Si SD ∪ OBS ∪ { ¬ AB(c_1), ..., ¬ AB(c_k) } est inconsistant pour un certain ensemble de composants {c_1, c_2, ..., c_k }, alors ce dernier est appelé **ensemble de conflits.**

Théorème
Un ensemble E de composants est un diagnostic pour (SD, Composants, OBS) si et seulement si E est un 'Hitting Set' minimal pour la collection des ensembles de conflits du système (SD, Composants, OBS).
Les ensembles 'Hitting Set' minimaux sont générés par un algorithme qui construit un arbre T des 'hitting sets' et qui a les propriétés suivantes :

Soit C un ensemble de collection d'ensembles

1- Si C = ∅ alors la racine de l'arbre T est libellée par √, sinon elle est libellée par n'importe quel ensemble dans C.

2- Pour chaque nœud de T, H(n) est l'ensemble des labels des arcs dans le chemin depuis la racine jusqu'au nœud *n*. Le label L(n) d'un nœud *n* est n'importe quel ensemble ℜ de C tel que : ℜ ∩ H(n) = ∅.

Si cet ensemble existe, le label de *n* est ℜ. Pour tout élément *e* de ℜ, *n* a un successeur n_e qui lui est relié par un arc libellé par *e*. Si l'ensemble ℜ n'existe pas le label de *n* est √.

Dans [REIT 87] l'auteur déclare que les ensembles 'Hitting Set' minimaux sont les H(n) pour les nœuds ayant √ comme label.

Pour générer le plus petit arbre 'HS-tree' contenant seulement les ensembles 'Hitting Set', l'auteur propose un algorithme appelé 'Pruned HS-tree'.

2.3.2 Algorithme 'Pruned HS-tree'

On génère les H(n) d'un niveau N avant de passer au niveau N+1

- **Si** le nœud *n* a un label $S \in C$, et si *n'* est un nouveau nœud tel que $H(n') \cap S = \varnothing$, **alors** L(n') = S

- **Si** un nœud *n* a le label √ et le nœud *n'* est tel que $H(n) \subseteq H(n')$, **alors** fermer *n'* (un symbole **X** indique que le nœud est fermé dans l'arbre).
  Si le label de *n* est déjà généré et le nouveau nœud *n'* est tel que H(n) = H(n'), **alors** fermer *n'*.

- **Si** les nœuds *n* et *n'* ont les labels S et S' (respectivement) tels que $S' \subset S$, **alors** pour chaque $\lambda \in S - S'$, marquer l'arc n_λ comme un arc redondant.
  Un arc redondant et le sous-arbre auquel il est attaché peuvent être enlevés de l'arbre 'HS-tree' pour garder le fait que la 'Pruned HS-tree' contienne seulement les ensembles 'Hitting Set' minimaux pour l'ensemble C.

Exemple :

Étant donnés les ensembles de conflits suivants : { {XR1, AD1, AD2, OR1}, {XR1, XR2}, {XR1, AD2, OR1}}, la figure 2.2 présente l'arbre 'Pruned HS-tree' correspondant, générée à partir de l'algorithme ci-dessus.

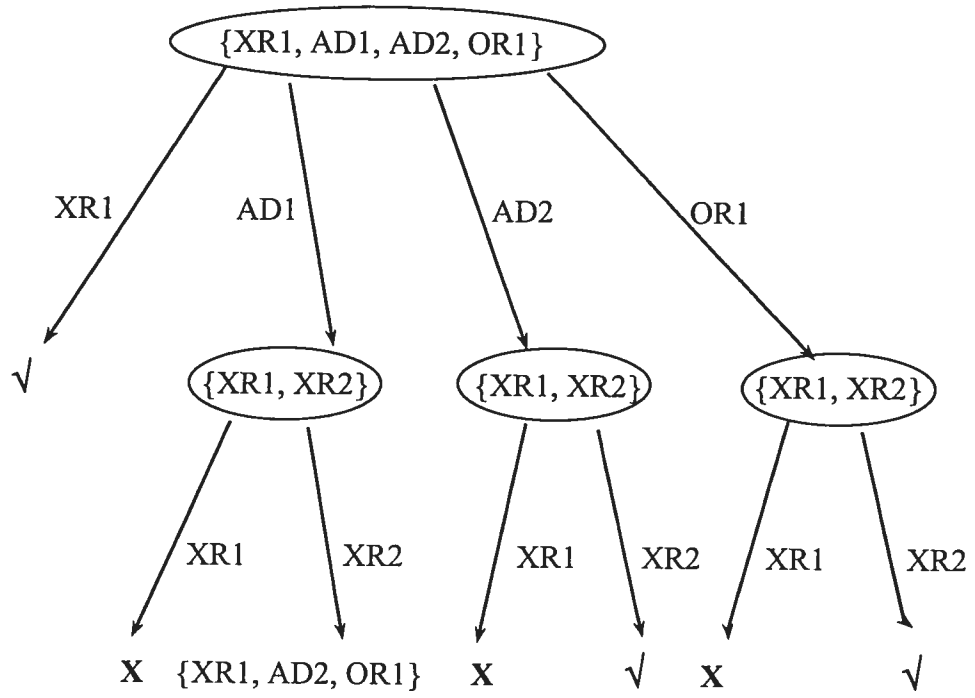

Figure 2.2 : Un exemple d'arbre 'pruned HS-tree'

2.3.3 Algorithme de diagnostic

L'algorithme procède en deux étapes :

- La première étape génère l'arbre T des ensembles 'Hitting-Set' pour les ensembles de conflits du système (SD, Composants, OBS) selon l'algorithme 'Pruned HS-tree'.
- La deuxième étape retourne les ensembles H(n), tel que *n* est un nœud de T étiqueté par √, qui constituent tous les diagnostics pour le système (SD, Composants, OBS).

Exemple :

D'après la figue 2.2, l'ensemble de tous les diagnostics pour le système {SD, {XR1, XR2, AD1, AD2, OR1}, OBS } est le suivant : { {XR1}, {AD2, XR2}, {OR1, XR2}}.

2.4 Algorithme général de diagnostic

Dans cette section, nous présentons l'algorithme proposé dans [GHED 93a] décrivant les étapes principales d'un processus de diagnostic. Le système sous diagnostic considéré est un système structuré constitué de plusieurs composants et il est décrit en termes de séquences d'entrées/sorties. Aucune technique particulière de génération des cas de test n'est considérée.

L'algorithme procède en six étapes :

Étape 1 : Génération des sorties attendues

Supposons que nous avons une suite de test (Ts) obtenue à partir d'une méthode de sélection des cas de test. La suite Ts est composée d'un ensemble de cas de test sous forme de symboles d'entrée. Par convention, on écrit $Ts = (Tc_1 \ldots. Tc_p)$ où chaque Tc_i est un cas de test.

Si un cas de test Tc_i est une suite de m_i entrées $I_{i,1}, I_{i,2}, \ldots, I_{i,mi}$; la séquence de sorties attendue est $O_i = O_{i,1}, \ldots., O_{i,mi}$ où Chaque $O_{i,j}$ est la sortie attendue correspondant à l'entrée $I_{i,j}$.

Étape 2 : Exécution des cas de test

C'est l'étape de l'application de la suite de test à l'implantation du système. Pour chaque cas de test Tc_i, une séquence de sortie correspondante $\hat{O}_i$ est observée sous la forme $\hat{O}_i = \hat{O}_{i,1}, \hat{O}_{i,2}, \ldots., \hat{O}_{i,mi}$.

Étape 3 : Génération des symptômes

Un générateur de symptômes compare les sorties observées aux sorties attendues pour chaque cas de test et identifie les symptômes. Chaque différence ($\hat{O}_{i,j} \neq O_{i,j}$) est un symptôme. La sortie fautive correspondant au symptôme est appelée *sortie symptôme.*

Étape 4 : Génération des ensembles de conflits

Pour chaque symptôme, on détermine l'ensemble de conflits correspondant. Un ensemble de conflits est défini comme étant l'ensemble des composants susceptibles de générer les sorties symptômes.

Étape 5 : Génération des candidats diagnostics

Les candidats diagnostics sont les composants suspects d'être fautifs. Ils doivent appartenir à chaque ensemble de conflits et doivent être consistants avec l'ensemble des observations.

Étape 6 : Discrimination entre les candidats

Dans cette étape, des tests diagnostics supplémentaires ou bien différents points d'observations peuvent être crées pour réduire l'espace des candidats diagnostics.

2.5 Conclusion

Dans ce chapitre, nous avons présenté deux méthodes de diagnostic existantes dans les domaines de l'intelligence artificielle et les circuits logiques. En particulier, nous avons détaillé les deux approches qui constituent la base de la littérature en matière de diagnostic, notamment l'approche de Reiter et celle de Davis. Ces dernières sont basées sur la structure et le comportement du système décrit par un modèle. Ensuite, nous avons détaillé les différentes étapes d'un processus de diagnostic pour les systèmes décrits par des séquences d'entrées/sorties comme ceux qui sont modélisés par les MEFs ou par les automates temporisées.

Dans le chapitre suivant, nous allons présenter les méthodes de diagnostic pour les protocoles de communication spécifiés par les MEFs et les MEFEs. Certaines de ces méthodes constitueront un point de départ pour le développement de notre méthode de diagnostic des systèmes temps réel.

Chapitre 3

Diagnostic des protocoles de communication

Durant les dernières décennies, la génération des séquences de test pour les protocoles de communication a été un point d'intérêt de plusieurs chercheurs. Plusieurs méthodes de sélection des séquences de test ont été développées pour tester la conformité des implantations de protocoles par rapport à leurs spécifications. Ces méthodes sont basées essentiellement sur le modèle des MEFs et elles permettent de détecter les fautes dans les transitions d'une IST. Néanmoins, l'application de ces séquences de test ne permet pas la localisation des fautes détectées ni le rétablissement du bon fonctionnement du système. Dans ce chapitre, nous allons présenter l'état de l'art de diagnostic basé sur les modèles formels de spécification. Nous nous intéressons surtout aux modèles et aux techniques qui sont utilisés dans le domaine des protocoles de communication. Plus précisément, nous aborderons le diagnostic basé sur le modèle des MEFs et le diagnostic fondé sur les MEFEs.

3.1 Approches de diagnostic des protocoles de communication

Après avoir détecté les fautes dans une IST d'un protocole de communication, il est important de localiser ces fautes. Deux approches sont utilisées dans ce sens :

- La première approche utilise les résultats de test et elle se base sur la spécification du système et les observations des traces de test pour expliquer les fautes détectées. Cette approche permet de trouver un ensemble de transitions suspectes. Dans le cas où le modèle de fautes est utilisé, d'autres séquences de test supplémentaires peuvent réduire cet ensemble et localiser les fautes. Cette approche n'exige pas que

toutes les transitions de la spécification soient exécutées lors de l'application des cas de test.

- La deuxième approche pour la localisation des fautes est celle qui construit, dans une première étape, toutes les machines possibles qui représentent le comportement observé et dont le nombre d'états ne dépasse pas une limite maximale. Puis, dans une deuxième étape, elle identifie la machine qui correspond à l'IST en utilisant des cas de test supplémentaires. Contrairement à la première, cette deuxième approche exige que toutes les transitions de la spécification soient couvertes par la suite de test initiale. Elle est plus complexe et plus coûteuse que la première approche, puisqu'elle ne fait pas la classification des fautes (fautes simples et fautes multiples).

3.2 Génération des cas de test pour les protocoles de communication

Dépendamment du formalisme utilisé pour définir la spécification d'un protocole, plusieurs techniques peuvent être utilisées pour la génération des séquences de test ou pour définir le processus de diagnostic. Dans plusieurs cas, une distinction est faite entre le flux de contrôle de la spécification et le flux de données.
La partie contrôle est souvent complétée par la partie des données pour la description des protocoles sous forme d'automates à états finis étendus par exemple.
Plusieurs chercheurs s'intéressent au test de cette partie de données complémentaires. Il s'agit d'une vérification des paramètres d'entrées, des paramètres de sorties et des variables locales en utilisant certaines formes d'analyse de flux et des données statiques. D'autres méthodes ont été développées aussi pour tester la partie donnée dans les systèmes logiciels d'une manière générale [MORE 90].

Concernant le test de flux de contrôle, plusieurs méthodes de sélection des cas de test [NAIT 81][SABN 88][FUJI 91][VUON 89] ont été développées dans le cas d'une spécification sous forme d'une MEF. Ces méthodes génèrent une suite de test, à partir de la spécification, pour déterminer si l'IST est conforme à cette spécification. Une suite de

test doit être relativement courte et doit couvrir le maximum de fautes que peut contenir l'implantation.

3.3 Le diagnostic basé sur les Machines à États Finis

Dans cette section, nous présentons un algorithme qui constitue une étape supplémentaire pour localiser les fautes détectées lors de la phase de test. Il s'agit de l'algorithme de diagnostic proposé dans [GHED 93a] qui permet, tout d'abord, de générer un ensemble de suspects expliquant les observations. Puis, de réduire le nombre de ces suspects en appliquant des séquences de test supplémentaires.

L'algorithme assure le diagnostic d'une faute simple (faute de sortie ou de transfert) dans une MEF. Un exemple illustratif est utilisé dans les différentes étapes proposées.

3.3.1 Le modèle des MEFs (déterministes)

Une machine à états finis peut être représentée par un tuple (S, I, Y, T, O)

- S est l'ensemble des états incluant l'état initial S_0.
- I est l'ensemble des symboles d'entrées.
- Y est l'ensemble des symboles de sorties incluant la sortie vide ('-').
- T est la fonction de transition d'états, $T: S \times I \longrightarrow S$.
- O est la fonction de sortie, $O: S \times I \longrightarrow Y$.

Un exemple de MEF est donné dans la figure suivante :

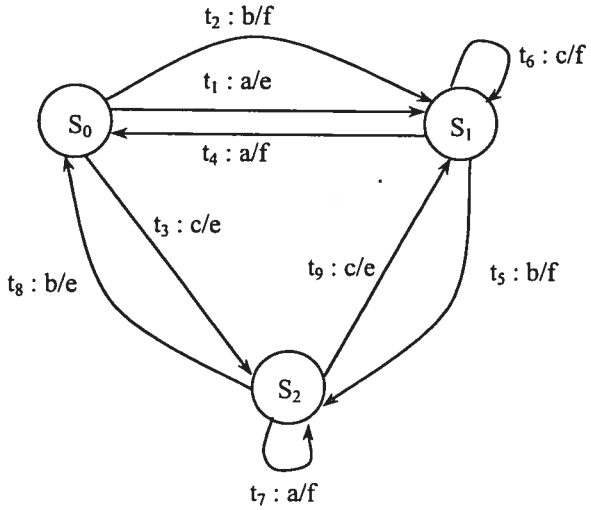

Figure 3.1 : Exemple de MEF

3.3.2 Modèle de fautes pour les MEFs

Le modèle de fautes pour les MEFs est basé sur les fautes de sorties des transitions et les fautes de transfert d'états de la machine.

On distingue essentiellement deux types de fautes dans une MEF :

a- Faute de sortie : On dit qu'une transition présente une faute de sortie, si pour un état donné et l'entrée correspondante, l'implantation produit une sortie différente de celle spécifiée par la fonction de sortie.

Une implantation présente une **faute de sortie simple** si une et seulement une de ses transitions présente une faute de sortie.

b- Faute de transfert : On dit qu'une transition présente une faute de transfert, si pour un état donné et une entrée correspondante, l'implantation transfère vers un état différent de celui spécifié par la fonction de transition d'états.

Une implantation présente une **faute de transfert simple** si une et seulement une de ses transitions présente une faute de transfert.

Nous supposons que le nombre d'états de l'implantation ne dépasse pas celui de la spécification.

3.3.3 Méthodes de sélection des cas de test pour les MEFs

Plusieurs méthodes de sélection de cas de test ont été développées pour les MEFs. Les méthodes les plus importantes sont les suivantes :

a- La méthode '**tour de transition**' [NAIT 81] détecte n'importe quelle faute de sortie dans l'absence d'une faute de transfert.

b- La méthode de séquence de distinction ou encore '**Ds-Method**' [GONE 70] et la méthode **UIO** [SABN 88] pour 'Unique Input Output' détectent les fautes de transfert (en supposant que le nombre d'états dans l'implantation et le même que dans la spécification).

c- Les méthodes W et Wp [FUJI 91] détectent en plus les fautes de transfert vers des états additionnels si le nombre de ces états est limité.

3.3.4 Diagnostic des fautes de sortie et de transfert dans les MEFs

Dans cette section, nous présentons l'algorithme de diagnostic proposé dans [GHED 93a] dans le cas où l'implantation et son modèle sont présentés par des MEFs déterministes. Les transitions de la machine à états finis sont considérées comme des composantes dans la description générale du système.

Cet algorithme est une version adaptée de l'algorithme général, présenté dans le chapitre 2 pour le diagnostic des transitions fautives d'une IST par rapport à son modèle de MEF déterministe.

Nous utilisons le modèle de fautes suivant :

l'IST peut avoir une faute de sortie ou bien une faute de transfert dans, au maximum, une de ses transitions.

Le but est d'identifier la transition fautive et le type de la faute (sortie ou transfert).

3.3.5 Définitions

La transition $t_{i,j}$ d'une spécification où le premier symptôme ($O_{i,j} \neq \hat{O}_{i,j}$) est apparu, lors de l'application de cas de test Tc_i, est appelée la **transition symptôme**.

Si la même transition symptôme est commune à tous les premiers symptômes des différents Tc_i, la transition est appelée l'**unique transition symptôme (uts)**. La sortie observée générée par l'uts est appelée l'**unique sortie symptôme (uss)**.

Pour montrer le déroulement de l'algorithme de diagnostic, un exemple est utilisé pour illustrer les différentes étapes. Nous supposons qu'il existe une implantation fautive I (figure 3.2) de la spécification de la figure 3.1.

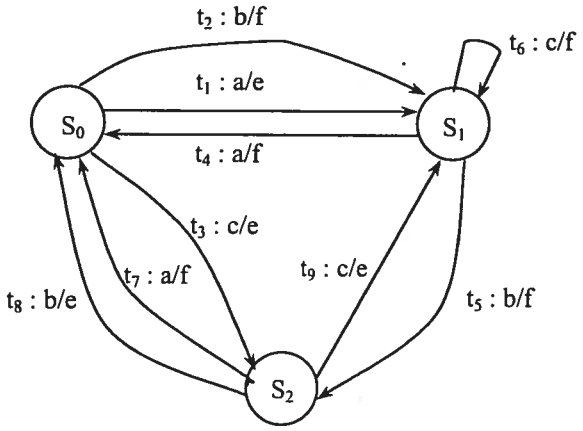

Figure 3.2 : Implantation fautive I de la MEF

3.3.6 Algorithme de diagnostic

Les différentes étapes de l'algorithme sont les suivantes :

Étape 1 à 3 (Génération des sorties attendues, exécution des cas de test et génération des symptômes).
Ces trois étapes correspondent aux trois premières étapes décrites dans l'algorithme de la section 2.4 .

Exemple : Nous supposons que la suite de test (Ts), générée à partir de la spécification de la figure 3.1, est donné comme suit :

Ts = (r,a,b ; r,b,c,a ; r,c,a,b ; r,c,a,c,b)

L'application de la suite Ts à la spécification et à l'implantation I génère les sorties attendues et les sorties observées illustrées dans le tableau suivant :

Tc_i.	Tc_1	Tc_2	Tc_3	Tc_4
Séquences d'entrées	r, a, b	r, b, c, a	r, c, a, b	r, c, a, c, b
Transitions parcourues	t_r, t_1, t_5	t_r, t_2, t_6, t_4	**t_r, t_3, t_7, t_8**	**t_r, t_3, t_7, t_9, t_5**
Sorties attendues	-, e, f	-, f, f, f	**-, e, f, e**	**-, e, f, e, f**
Sorties observées	-, e, f	-, f, f, f	**-, e, f, f**	**-, e, f, e, e**

Table 3.1 : Les cas de test et leurs séquences de sorties correspondantes

Nous supposons qu'il existe une transition t_r permettant, à tout moment, de ramener la spécification et l'implantation à l'état initial. Nous utilisons le symbole 'r' pour noter l'entrée de cette transition et le symbole '-' pour noter sa sortie vide.

Une différence entre les sorties attendues et les sorties observées est détectée pour les cas de tests Tc_3 et Tc_4. les symptômes correspondants sont les suivants :

$Symp_3$ ($O_{3,3} \neq \hat{O}_{3,3}$) avec la transition t_8 comme transition symptôme.
$Symp_4$ ($O_{4,4} \neq \hat{O}_{4,4}$) avec la transition t_5 comme transition symptôme.

Pour le reste des étapes de l'algorithme nous supposons que l'implantation a une faute simple (faute de sortie ou faute de transfert).

Étape 4 : Pour chaque symptôme détecté lors de l'application d'un cas de test, un ensemble de conflits est formé des transitions exécutées d'après la spécification. Aucune transition exécutée après l'observation du premier symptôme n'est incluse dans cet ensemble. La transition réellement fautive est incluse dans chaque ensemble de conflits.

Exemple : Les 2 ensembles de conflits correspondant aux 2 symptômes détectés sont :

Conf $_3$ = (t_3, t_7, t_8).
Conf $_4$ = (t_3, t_7, t_9, t_5).

Étape 5A : **L'Ensemble des Candidats Initiaux (ECI)** sera formé de l'intersection de tous les ensembles de conflits. Chaque transition t_k dans l'ECI représente une **transition candidate** pour une faute de transfert ou une faute de sortie qui peut expliquer tous les symptômes.

Exemple :
L'ECI correspondant aux deux ensembles de conflits générés est le suivant :
ECI = (t_3, t_7).

Étape 5B : **Ensemble des Candidats Finaux (ECF)**
S'il existe une **transition symptôme unique (tsu)**, elle sera incluse dans l'ECI. Dans ce cas, l'ECI est divisé en deux ensembles. un **Ensemble de Transitions Symptômes Uniques (ETSU)**, qui contient la tsu, et un **Ensemble de Candidats Finaux (ECF)** qui contient le reste des transitions dans l'ECI. Sinon, l'ensemble ECI constituera l'ECF.

Si une transition tsu existe, nous considérons chaque cas de test de Ts qui contient, au moins, une transition tsu. Si les sorties observées correspondant à la tsu sont toutes égales à la **sortie symptôme unique (ssu)** et pour les transitions restantes, les sorties attendues et les sorties observées sont égales, alors la tsu est un candidat diagnostic pour une faute de sortie.

Pour chaque transition $t_k \in$ ECF, nous considérons l'ensemble des états auxquels t_k peut transférer à l'exception de l'état suivant t_k dans la spécification. Pour chaque état s considéré, et sous l'hypothèse que s est l'état suivant t_k, si les sorties déjà observées et les sorties attendues de cette nouvelle spécification sont égales pour toutes les transitions des différents cas de test, alors s est ajouté à un ensemble dit **ensemble d'états finaux de t_k** noté par **EtatsFinaux(t_k)**.

La transition réellement fautive sera forcément détectée lors de cette étape puisque toutes les transitions dans l'intersection des ensembles de conflits sont examinées pour toutes les fautes possibles.

Exemple :

Puisqu'il n'y a pas de tsu dans l'ECI de notre exemple, il n'y a pas de faute de sortie et les éléments de l'ECI constitue l'ensemble des candidats finaux ECF suivant :

ECF = (t_3, t_7).

Les ensembles des états finaux pour les deux transitions de l'ECF sont les suivants :

EtatsFinaux(t_3) = ∅.

EtatsFinaux(t_7) = (S_0, S_1).

Étape 5C : Identification des candidats plausibles et génération des diagnostics

Dans cette étape, nous éliminons toutes les transitions t_k correctes de l'ECF. Ce sont les transitions qui ont leurs ensembles d'états finaux (EtatsFinaux(t_k)) vides. L'ensemble résultant est appelé Ensemble des Candidats Diagnostics (ECD). Pour chaque transition $t_k \in$ ECD et pour chaque état $S_{k,i} \in$ EtatsFinaux(t_k), un 'diagnostic' prétend que la transition t_k transfert à $S_{k,i}$ au lieu de l'état suivant dans la spécification. Un autre diagnostic prétend que la tsu présente une faute de sortie si l'ETSU ≠ ∅.

Exemple :

Les transitions ayant un ensemble d'états finaux vide sont correctes et elles sont alors exclues de l'ECF. L'ensemble des candidats diagnostic (ECD) contient les transitions suspectes pour la faute de transfert. Aucune transition n'est suspecte pour la faute de sortie puisque l'ETSU=∅. l'ECD obtenu est le suivant : **ECD = {t_7}**

Pour chaque état de l'ensemble EtatsFinaux(t_7), nous pouvons conclure un diagnostic qui explique les symptômes observés :

Diag1: t_7 transfére à l'état S_0 au lieu de S_2.
Diag2: t_7 transfére à l'état S_1 au lieu de S_2.

Étape 6 : Génération des tests diagnostics additionnels

Dépendamment des résultats obtenus lors des étapes précédentes, plusieurs cas peuvent se présenter :

Cas1 : L'ETSU contient la tsu et CD=∅. Dans ce cas la tsu est la transition réellement fautive. Celle-ci présente une faute de sortie et nous n'avons plus besoin de tests diagnostics additionnels.

Cas2 : L'ETSU est vide et CD est un singleton avec un seul état final correspondant. Dans ce cas la transition de CD présente une faute de transfert et aucun test diagnostic n'est nécessaire.

Cas3 : L'ETSU est vide et CD est un singleton $\{t_k\}$ avec un ensemble d'états finaux contenant plus qu'un élément, ou bien CD contient plusieurs transitions. Dans ce cas, chaque transition de CD peut avoir une faute de transfert. D'autres cas de test supplémentaires sont nécessaires pour identifier la transition réellement fautive et l'état auquel elle transfére.

Pour générer ces cas de test, l'algorithme proposé est le suivant :

Pour chaque transition t_k de l'ECD, d'autres cas de test doivent pouvoir distinguer entre les différents états $S_{k,i}$ de l'ensemble EtatsFinaux(t_k). Ainsi un ensemble de caractérisation $W_{k,i}$ doit être calculé pour chaque état $S_{k,i}$. Cet ensemble est formé d'une séquence d'entrées telle que :
Si cette séquence est appliquée à la machine dans l'état $S_{k,i}$, la séquence de sorties générée sera différente de celle générée si nous appliquons la même séquence à la machine dans un autre état $S_{k,j}$.

Chaque cas de test supplémentaire est une concaténation d'une séquence d'entrées dite *séquence de transfert*, pouvant amener la machine depuis l'état S_0 à l'état initial de t_k, de l'entrée de t_k et de la séquence d'entrées $W_{k,i}$.

Exemple :
Les séquences de distinction $W_{7,0}$, $W_{7,1}$ caractérisant les états S_0 et S_1 peuvent être la même entrée 'a'. Cette dernière appliquée à la spécification à l'état S_0 et S_1 génère respectivement les sorties 'e' et 'f' ; ce qui permet de distinguer ces deux états. Une séquence de transfert possible permettant d'amener la machine à l'état initial de t_7 peut être la séquence 'r, c'. Par conséquent, un cas de test supplémentaire est 'r, c, a, a' tel que le premier symbole 'a' est l'entrée de la transition t_7.

Ce cas de test appliqué à l'implantation (figure 3.2) produit la séquence de sorties '-, e, f, e'. Ce qui permet de conclure que la transition t_7 transfert bien à l'état S_0 au lieu de S_2.

3.3.7 Conclusion

Il est important de noter que l'algorithme détaillé (section 3.3.4) dépend de la première étape, quand différentes méthodes de sélection des séquences de test peuvent être utilisées pour générer la suite de test initiale. Pour une méthode de sélection de test incomplète, telle que la méthode 'Transition Tour', nous ne pouvons pas garantir que les fautes vont être détectées. Dans ce cas, aucun diagnostic ne pourra être généré et les étapes 5 à 7 ne seront pas atteintes. Par conséquent, la tâche de diagnostic devient difficile.

Par contre, pour une suite de test plus complète, telle que celle générée par la méthode UIOv ou Wp, nous pouvons garantir que les fautes vont être détectées, mais elle ne seront pas nécessairement localisées. Dans ce cas, plusieurs diagnostics peuvent être générés à l'étape 5, puis des cas de test additionnels peuvent être aussi générés à l'étape 7.

La méthode de diagnostic que nous venons de décrire a été étendue dans [BOUM 00] au cas des MEFEs.

3.4 Le diagnostic basé sur les Machines à États Finis Étendues (MEFEs)

La limitation majeure des MEFs est le problème d'explosion d'états ou d'explosion combinatoire. Afin de décrire un système qui utilise des compteurs ou des structures de données plus complexes, il faut utiliser un très grand nombre d'états.

En plus, les techniques de test basées sur le modèle de MEF sont utilisées pour tester l'aspect contrôle d'une IST. Cependant, le flux de données est plus facilement diagnostiqué s'il est représenté sous forme de MEFE.

Dans cette section, nous présentons l'algorithme proposé dans [BOUM 00] pour le diagnostic de flux de données pour les systèmes modélisés par les MEFEs, ainsi que le modèle de fautes utilisé.

3.4.1 Le Modèle des machines à états finis étendues

Le modèle de MEFE décrit un module par une MEF qui est étendue par ce qui suit :

- Les données d'entrées et de sorties ont des paramètres typés.
- Le module a un certain nombre de variables locales typées dites variables d'état ou de contexte.
- A chaque transition est associé un prédicat qui dépend des paramètres d'entrées et des valeurs des variables d'états.
- Une action qui peut modifier les variables d'états est réalisée quand la transition est exécutée.

Un modèle de MEFE peut être représenté par un graphe qui sera l'extension du graphe de la MEF par l'adjonction des trois fonctions F_1, F_2 et F_3 telles que :

F_1: I x X $\longrightarrow$ P, F_2: I x X $\longrightarrow$ O et F_3: I x X $\longrightarrow$ X

I, O, X, P : représentent respectivement l'ensemble des paramètres d'entrées, l'ensemble des paramètres de sorties, les variables d'états et les prédicats des transitions.

- F_1 définit la vérification du prédicat par les paramètres d'entrées et les variables d'états. Notons que pour les transitions spontanées I = ∅, et pour les transitions sans prédicat, P = Vrai.
- F_2 décrit l'influence des paramètres d'entrée et des variables de contexte sur les variables de sorties
- F_3 définit les nouvelles valeurs des variables d'états après l'exécution d'une transition

3.4.2 Définitions

- Une transition *t* a un usage d'affectation (**A-Use**) par rapport à une variable x si x apparaît dans le membre gauche d'une instruction d'affectation de t.
- Si x apparaît dans la liste d'entrées d'une transition t, x est dite un usage d'entrée (**I- use**) dans la transition t.
- Quand une variable apparaît dans la partie condition d'une transition t, x est usage de prédicat (**P-use**).
- x est un usage de calcul (**C-Use**) dans une transition t si x apparaît dans une primitive de sortie ou dans la partie droite d'une instruction d'affectation de t
- x est un usage de définition si x est un A-use ou un I-use
- Un **usage global** d'une variable x est un usage de x dont la définition apparaît dans d'autres transitions, sinon c'est un usage local
- Une **définition globale** de x c'est une définition de x pour laquelle il existe un usage global dans une autre transition, sinon c'est une définition locale.
- un chemin $(t_1,t_2,\ldots t_k, t_n)$ est dit un chemin **def-clear** par rapport à une variable x si les transitions $t_2 \ldots t_k$ ne contiennent aucune définition de x.
- un chemin $(t_1,\ldots..,t_n)$ est dit un chemin **def-use** par rapport à une variable x si $x \in def(t_1)$, $x \in$ C-Use(t_n) ou $x \in$ P-Use(t_n) et $(t_1,\ldots\ldots,t_n)$ est un chemin def-clear par rapport à x.

Dans l'exemple de la MEFE présenté dans la figure 3.3 (page 29), la transition (5) contient une définition, deux usages du paramètre d'entrée 'ISDU' et une définition de la variable d'état 'olddata'. La transition (7) contient des usages de définition et de prédicat des variables 'Num' et 'counter', un usage de calcul de 'number' et 'counter' et un usage de prédicat de 'number'.

Afin de pouvoir tester le flux de données dans les systèmes modélisés par les MEFEs, un choix de critère doit être fait. Plus le critère est fort, plus le système est vérifié et testé d'une manière efficace. Le critère le plus fort est de parcourir *tous les chemins*. Cependant ce critère est trop coûteux et n'est pas pratique puisque les boucles dans la spécification peuvent produire une infinité de chemins. Tester uniquement les chemins contenant des relations de flux de données est une solution raisonnable, solution qui utilise comme critère une relation de dépendance entre les transitions comme la relation *définition-usage* ou def-use.

Deux types de dépendance sont définis entre deux transitions t_i et t_j :

t_i est dépendante par ses données (ou par le contrôle) de la transition t_j, s'il existe une variable x telle que :

- t_i contient une définition globale de x.
- t_j contient un usage de calcul (ou un usage de prédicat de x).
- il existe un chemin def-clear de t_i à t_j par rapport à x.

Ainsi, pour tester le flux de données, nous générons tous les chemins def-use qui permettent de tester les dépendances qui existent dans le système entre les transitions (dépendances de données et de contrôle).

Par exemple dans le modèle de la figure 3.3, la transition (7) est dépendante par les données et le contrôle de la transition (6) par rapport aux variables 'number' et 'counter'. Le chemin (6, 5, 7) est un chemin def-clear pour ces deux variables.

Il faut bien noter qu'il existe une différence entre le critère def-use et le modèle de fautes [BOUR 99]. Dans le cas de critère def-use, l'objectif est de générer des cas de test qui parcourent les chemins correspondant au critère. Tester ces chemins ne garantit pas la détection des fautes existantes à cause des valeurs des variables qui doivent être bien

choisies. Dans le cas ou les bonnes valeurs sont choisies, le critère def-use est comparable à un modèle de fautes.

3.4.3 Modèle de fautes pour les MEFEs

Le modèle de fautes que nous allons présenter s'inspire du critère def-use que nous avons présenté dans le paragraphe précédent. trois types de fautes y sont considérés :

a- Faute de définition : On dit qu'une transition a une faute de définition, si la ou les variables d'états $x \in X$ qui lui sont associées reçoivent des valeurs différentes de celles définies dans le modèle du système. Ce qui peut être expliqué par une erreur au niveau de l'exécution de l'action associée à la transition.

Dans ce cas de faute, le symptôme apparaîtra plus tard, après l'exécution de la transition dont l'action est erronée. On parle ici d'une ***faute latente***. Elle sera éventuellement le résultat d'une insatisfaction de la condition d'une transition qui va être exécutée ou encore la satisfaction de la condition d'une transition autre que celle prévue par le modèle.

b- Faute de transfert : On dit qu'une transition a une faute de transfert, si pour un état donné et pour l'entrée correspondante, elle transfère vers un état autre que celui spécifié dans le modèle du système. Une faute de transfert est due au fait que la condition de garde est non satisfaite. Deux types d'erreurs sont possibles :

- Une erreur au niveau de la condition elle-même.
- Une erreur de définition.

Pour les fautes de transfert, le symptôme apparaît immédiatement après l'exécution de la transition induisant une sortie différente que celle définie dans le modèle.

c- Faute de paramètres de sortie : On dit qu'une transition a une faute de paramètre de sortie, si pour un état donné et l'entrée correspondante, elle produit une sortie avec une valeur de paramètre autre que celle définie par le modèle.

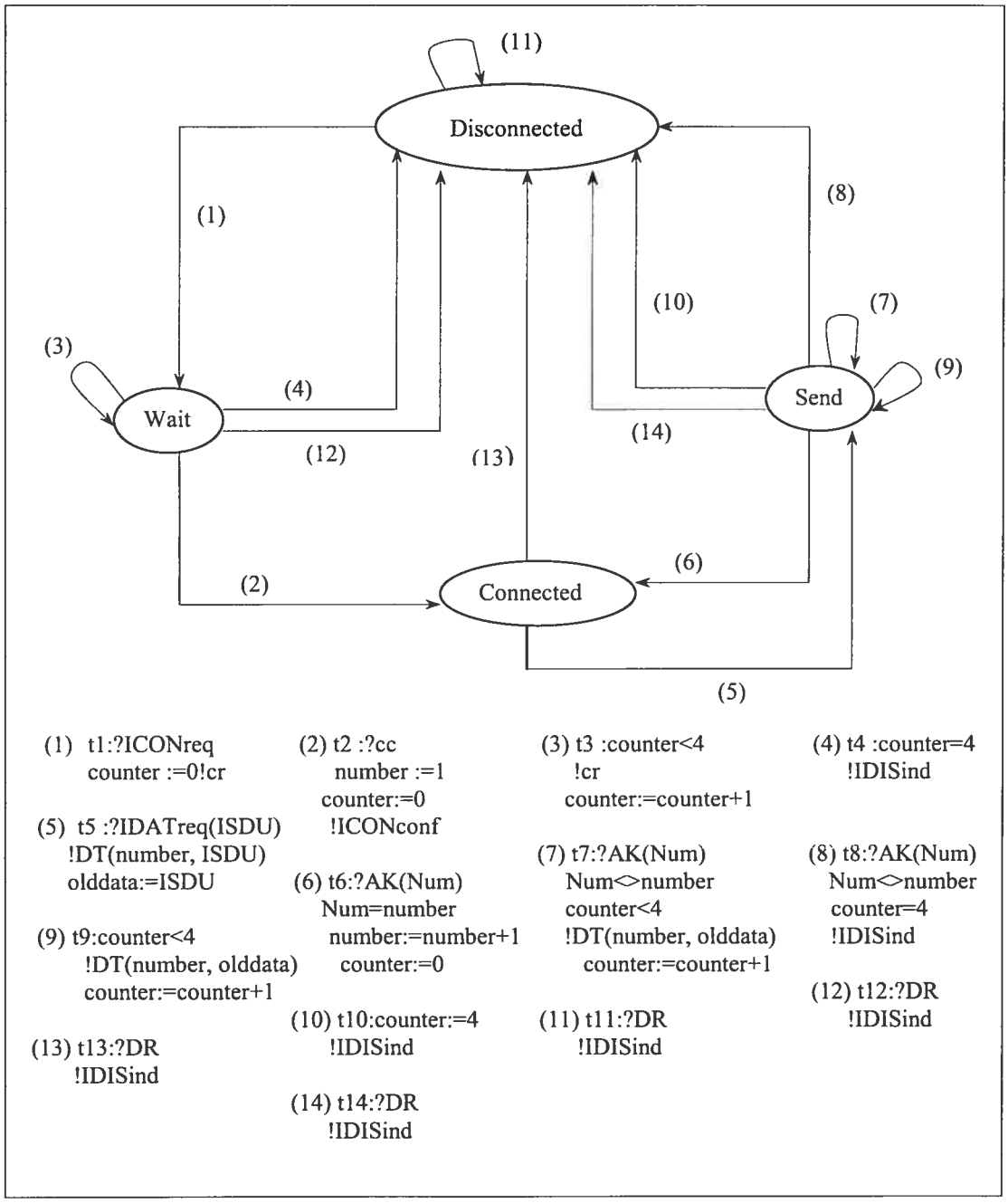

Figure 3.3 : Exemple de MEFE

3.4.4 Algorithme de diagnostic

Cet algorithme est développé dans [BOUM 00] en s'inspirant de l'algorithme général de diagnostic [GHED 93a] déjà présenté dans la section 2.4 (page 13). Le comportement du système sous diagnostic est décrit sous forme d'entrées et de sorties, et l'application d'une séquence d'entrées au système génère les sorties correspondantes.

Hypothèses

- Pour chaque faute dans l'IST, il existe un cas de test qui détectera cette faute.
- Pour chaque faute dans l'IST, il existe un cas de test qui l'atteint directement.

On dit qu'une faute est directement atteinte par un cas de test, si toutes les fautes qui sont antérieures à cette faute ne sont pas des fautes de transfert.

Dans le cas d'une faute de transfert ou une faute de paramètre de sortie, le symptôme apparaîtra immédiatement après l'exécution de la transition erronée ; par contre dans le cas d'une faute de définition, le symptôme apparaîtra plus tard. Nous supposons, dans tous les cas que la faute sera détectée.

Étapes 1 à 3 : Génération des sorties attendues, exécution des cas de tests et génération des symptômes

Ces trois étapes sont les mêmes que celles décrites dans l'algorithme général (voir section 2.4).

Étape 4 : Génération du graphe de dépendance des transitions

Comme nous l'avons déjà mentionné, les transitions d'une MEFE sont dépendantes les unes des autres. Cette dépendance est due aux variables de ces transitions.

Pour pouvoir analyser les symptômes et générer les candidats, un graphe de dépendance des transitions est nécessaire.

Ce graphe fournit, à partir d'une MEFE, les dépendances (des données et de contrôle) définition-usage des variables x des transitions.

L'idée principale de cette étape est de déterminer pour chaque usage de variables d'une transition dans le modèle, la ou les transitions qui contiennent les définitions de ces variables puisqu'une erreur de contrôle (usage de prédicat) ou de paramètre de sortie (usage de calcul) peut être due à une mauvaise définition des variables sur lesquelles portent ces usages.
Ceci étant fait dans le but d'avoir des hypothèses sur les transitions fautives.

Étape 5 : Construction des ensembles de candidats plausibles

A partir des symptômes générés lors de l'étape 3 et du graphe de dépendance des transitions, nous construisons un ensemble de candidats qui pourront expliquer les symptômes reçus. Un ensemble d'hypothèses est généré pour les transitions suspectées fautives.

Si nous avons m symptômes, nous les considérons dans l'ordre. Tous les symptômes avant une faute de transfert (y compris les fautes de définition qui se manifestent comme des fautes de contrôles) n'étant pas considérés. Nous supposons qu'ils ne sont pas directement atteints par ce cas de test.

Dans le cas d'une erreur de contrôle, nous émettons deux hypothèses. La première concernant la condition de garde de la transition elle-même, suspectant sa transcription erronée. La deuxième suppose que les variables de la condition sont mal définies. Dans ce dernier cas, l'analyseur doit monter aux transitions définissant la variable pour émettre les hypothèses à propos des variables de ces transitions. C'est là nous utilisons les chemins def-usage du graphe de dépendance des transitions.
Dans le cas d'une erreur de paramètre de sortie, l'analyseur émet des hypothèses sur les transitions qui définissent les valeurs de ces paramètres.

Les hypothèses de tous les cas de tests sont réunies afin d'avoir des hypothèses globales sur l'IST.

Étape 6 : Discrimination entre les candidats et génération des diagnostics

Cette étape offre un moyen de réduire l'ensemble des candidats potentiels générés lors de la dernière étape. En effet, nous utilisons dans cette étape la technique de ***mutation*** pour

vérifier si le candidat réagit de la même façon que l'implantation lors de l'application de la suite de test. Un mutant est construit à partir de la spécification en injectant des erreurs aux variables des instructions dans les transitions symptômes.

Dans [BOUM 00] l'auteur a défini un ensemble de règles qui associent à chaque type de faute un ensemble d'opérations de modifications.

Pour les actions de transitions, les opérations seront d'affecter à la variable d'état un ensemble de valeurs distinctes (la plus petite valeur possible, la plus grande valeur possible de l'intervalle, etc.)

Pour les prédicats et les conditions associées aux transitions, les opérations à effectuer seraient de remettre la condition à faux ou d'attribuer les valeurs limites des intervalles définissant les variables ; ou même toutes les valeurs de l'intervalle si celui ci est réduit.
Pour les variables dont les valeurs apparaissent dans les paramètres de sorties des transitions, il faut faire une exécution symbolique sur l'IST pour réduire les valeurs des données que nous supposons erronées.

3.5 Conclusion

Dans ce chapitre, nous avons présenté les méthodes de diagnostic des implantations représentées par des modèles. En particulier, nous avons détaillé deux algorithmes de diagnostic basés sur les MEFs et les MEFEs. Ces algorithmes supposent que la spécification du système, l'IST et la suite de test initiale sont disponibles.

D'après l'analyse des différentes méthodes de diagnostic, nous pouvons remarquer qu'il existe une forte dépendance entre les diagnostics générés et les tests additionnels, d'une part, et la suite de tests initiale d'autre part. En d'autre terme, si une suite de test initiale est suffisamment complète pour détecter les fautes existantes dans une IST, alors moins de diagnostics vont être générés et par la suite, moins de tests supplémentaires seront nécessaires. Sinon, un grand nombre de diagnostics sera généré, et par conséquent, plus de cas de test additionnels seront à sélectionner pour les distinguer.

Dans le prochain chapitre, nous étudierons les différentes méthodes de test des systèmes temps réel pour choisir celle que nous allons utiliser pour générer les cas de test dans notre approche de diagnostic. Le choix d'une telle méthode sera fondé sur sa complétude et son degré de couverture des fautes tout en considérant la complexité de l'algorithme global de diagnostic.

Chapitre 4

Choix de la méthode de test

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 les travaux qui ont été réalisés pour générer des cas de test temporisé et nous discutons les lacunes de chaque méthode. Ceci va guider notre choix de la méthode à utiliser dans notre approche de diagnostic.

4.1 Génération de tests à partir des formules logiques

Cette approche [MAND 95] 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*, référant 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, cette dernière est décomposée d'une manière hiérarchique en des formules atomiques ne contenant que des *événements*. Un événement est un couple (L, t) formé du littéral L et de l'instant t. Un ensemble d'événements qui ne contient pas de contradiction (c'est-à-dire deux événements (L, t) et ($\neg L$, t) en même temps) est appelé une *histoire*. Chaque feuille ne contenant aucune contradiction dans l'arbre de décomposition d'une formule F est dite complète si elle contient une valeur de vérité unique pour chaque prédicat de la formule en 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 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 une formule 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 retreinte des systèmes temps réel (ceux qui sont spécifiés à l'aide d'une seule horloge).

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

Cette méthode [LIUG 93] génère des cas de test temporisés à partir d'une MEF avec des temporisateurs et des compteurs. Elle utilise la méthode Wp [FIJU 91] en procédant comme suit. Premièrement, le comportement de chaque temporisateur et celui de chaque compteur sont modélisés par des MEFs. Ensuite, toutes 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 qui correspondent au déclenchement, l'arrêt ou l'expiration du temporisateur et au passage de la durée pour laquelle le temporisateur est activé.

- Un ensemble de transitions entre les états suite à des 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 représentant respectivement la remise à zéro du compteur et son incrémentation.
- Un ensemble de transitions entre les états suite à des 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 Génération de tests à partir d'un graphe de contraintes

La spécification utilisée dans cette approche [CLAR 97][CLAR 96] est donnée sous forme d'un Graphe de Contraintes noté 'GC'. À partir du modèle de fautes, les auteurs définissent certains *critères de test* et génèrent des cas de test qui les satisfont. Ces critères, illustrés dans la Figure 4.1, ressemblent en quelque 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 (c'est-à-dire lorsque le critère C_1 est satisfait le critère C_2 l'est aussi). L'inclusion marquée par un astérisque n'est vraie que dans des cas particuliers.

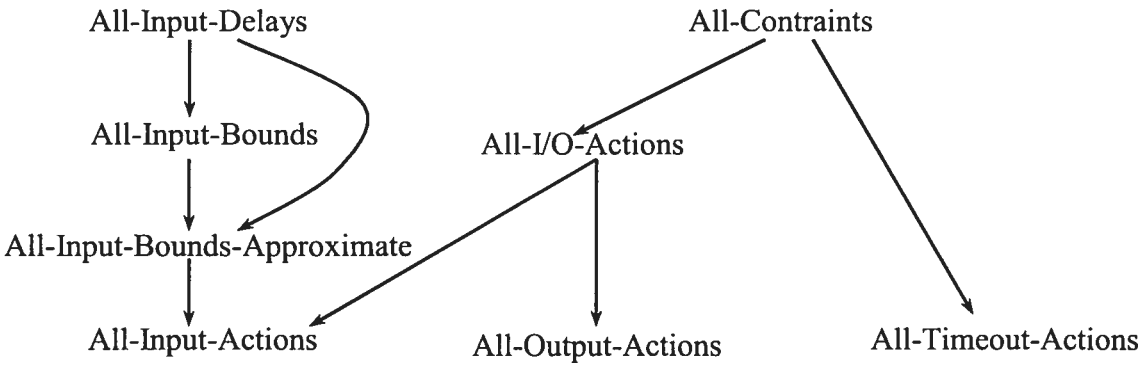

Figure 4.1 : Critères de test d'un graphe de contraintes

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-Appraximate* est défini et consiste à tester la 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/0-Actions* et *All-Timeout-Actions* sont satisfaits.

L'approche proposée a deux limitations. La première concerne le modèle (c'est à dire, le GC) utilisé pour la génération des cas de test. Le GC est un modèle particulier ne permettant de décrire que les délais minimum et maximum entre deux événements (entrée/sortie) successifs. La deuxième limitation est liée à la couverture des fautes susceptibles d'exister dans l'implantation sous test. Les cas de test satisfaisant les critères de la Figure 4.1 ne peuvent pas garantir une couverture complète de fautes.

4.4 Génération de tests à partir d'un automate temporisé

Springintveld *et al.* [SPRI 01] présentent un cadre théorique pour générer des cas de test à partir d'un automate temporisé (AT). Ils proposent une adaptation de la méthode W

[CHOW 78] pour prendre en considération l'aspect temporel d'un système temps réel. La relation de conformité utilisée pour juger la conformité de l'implantation par rapport à sa spécification est la bissimulation [MILN 80][MILN 89].

L'approche consiste en plusieurs transformations. Tout d'abord, on calcule le nombre de régions de l'AT résultant de la composition de l'automate de l'implantation et celui de la spécification. Ceci a pour but de déduire la longueur maximale de la trace de distinction des états. Ensuite, on construit un sous-automate du graphe des régions, dit *automate de grille,* en utilisant un *"mapping" uniforme* [CERA 92a] [CERA 92b] caractérisant la relation entre les interprétations d'horloges. L'automate résultat est enfin utilisé pour générer des cas de test en appliquant la méthode W.

L'automate de grille est un système de transitions étiquetées résumant le comportement du graphe de régions nécessaire à un test exhaustif du système. Chaque état de cet automate a une transition de délai sortante de durée 2^n, où n est la longueur de la trace de distinction des états dans le graphe de régions de la composition de l'automate de l'implantation et celui de la spécification. Dans le pire cas, n est égal au nombre de régions d'horloges dans ce graphe. La valeur 2^n est choisie de telle sorte que les cas de test générés puissent, sous certaines hypothèses, détecter toutes les erreurs possibles dans l'implantation.

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 bissimulation. En fait, toute implantation, passant avec succès la suite de test générée, est déclarée *bissimilaire* 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. Quand 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 [ALUR 94] .

4.5 Génération de tests à partir des Automates à Entrée Sorties Temporisées

Les méthodes précédentes souffrent de certains problèmes tels que la restriction des modèles utilisés et/ou la complexité et la non-applicabilité des approches proposées. Ceci nous a amené à considérer une méthode qui tend vers la pratique pour tester les systèmes temps réel, tout en se basant sur un modèle adéquat et général de spécification. Cette méthode, dite la méthode Wp temporisée [ENNO 01][ENNO 02], a aussi l'avantage d'avoir une couverture complète de fautes pour une granularité donnée. Le modèle des Automates à Entrées Sorties Temporisées (AESTs) est utilisé comme base pour générer des cas de test temporisés. Ce modèle est une variante des automates temporisées d'Alur et Dill où l'ensemble d'alphabet est subdivisé en un ensemble d'entrées 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 4.1 : *AEST*
Un automate à entrées sorties temporisées (AEST) est un quintuplet (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 $[0, Cx] \cup \{\infty\}$, où Cx est la plus grande constante entière utilisée dans les contraintes sur l'horloge x.

- $T \subseteq L \times L \times \{I \cup O\} \times 2^C \times \Phi(C_A)$ est un ensemble fini de transitions.

Un tuple $(l, l', a, \lambda, G) \in T$, noté par $l \xrightarrow{a,\lambda,G} l'$, représente une transition de l'emplacement l à l'emplacement l' sur l'événement a, Le sous ensemble $\lambda \subseteq C$ 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, formé à partir des formules logiques $x\ op\ m$ où $x \in C$, $op \in \{<, \leq, =, >, \geq\}$ et $m \in N$.

Chaque horloge $x \in C$ prend ses valeurs dans **R**, croit de manière synchrone par rapport à une horloge globale et mesure le temps écoulé depuis sa dernière réinitialisation.
Elle n'est pertinente qu'au-dessous de la constante Cx. C'est pourquoi, toutes les valeurs supérieures à Cx sont présentées par ∞ et nous écrivons : $\forall \varepsilon > 0,\ \forall x \in C,\ Cx + \varepsilon = \infty$.

Pour définir le modèle sémantique d'un AEST, nous définissons 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 $\mathbf{R}^{\geq 0} \cup \{+\infty\}$. L'ensemble des interprétations d'horloges est noté par $V(C)$. Une interprétation d'horloges ν satisfait une contrainte temporelle G, et nous le notons $\nu \models G$, si et seulement si G est évaluée à vrai sous ν.

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

Un état d'un AEST A est un couple (l, ν), où $l \in L$ et $\nu \in V(C)$. L'état initial de A est le couple (l_0, ν_0), où l_0 est l'emplacement initial de l'AEST et ν_0 est une interprétation d'horloges tel que $\nu_0(x) = 0$ pour toute horloge $x \in C$. L'ensemble des états de L'AEST est noté par S.
Nous distinguons deux types de transitions dans un AEST :

- Les *transitions de délai* : l'automate change d'état à chaque écoulement du temps. Les transitions de ce type ont la propriété d'additivité suivante :
  $s \xrightarrow{\varepsilon(d+d')} s' \Leftrightarrow \exists\, s'' \in S : s \xrightarrow{\varepsilon(d)} s'' \wedge s'' \xrightarrow{\varepsilon(d')} s'$ (ε signifie pas d'action).
- Les *transitions explicites* : à partir d'un état (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 $s \xrightarrow{a} s'$, où $a \in I \cup O$.

La figure 4.2 montre un exemple d'AEST avec deux horloges x et y. L'automate a deux emplacements l_0 (l'emplacement initial) et l_1 ainsi que trois transitions entre ces emplacements. Par exemple la transition $l_1 \xrightarrow{?In,\{x,y\},x<1\&y<1} l_0$ est tirable sur l'entrée *?In* lorsque les valeurs des horloges x et y sont inférieures à 1, remet à zéro x et y et ramène la machine à son emplacement initial. Le domaine temporel des deux horloges est le même et il est égal à [0, 1] $\cup \{\infty\}$.

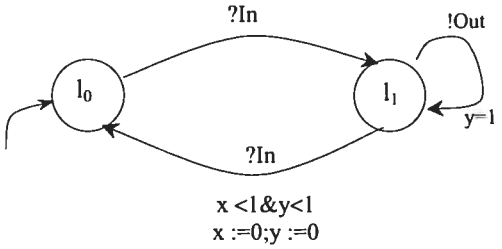

Figure 4.2 : Exemple d'AEST

Le modèle sémantique d'un AEST est donné par un système de transitions étiquetées temporisé $S_t = (S, I \cup O \cup \mathbf{R}^{\geq 0}, \rightarrow, (l_0, v_0))$ dont l'ensemble d'états est S, l'ensemble des alphabets est l'union de l'ensemble $\{I \cup O\}$ et l'ensemble de délai ($\mathbf{R}^{\geq 0}$) et la relation de transition inclut les deux types de transitions cités précédemment. 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, nous définissons une relation d'équivalence [ALUR 94] sur l'ensemble d'interprétations d'horloges *V(C)* de la manière suivante.

Définition 4.2 : *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 $[0, Cx] \cup \{\infty\}$. *Deux interprétations d'horloges* ν *et* ν' *sont dites équivalentes, que l'on note* $\nu \sim \nu'$, *si et seulement si* :

- $\forall x \in C, \lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$.
- $\forall x, y \in C \mid ((\nu(x) \neq \infty) \wedge (\nu(y) \neq \infty))$, $(fract(\nu(x)) \leq fract(\nu(y)) \Leftrightarrow fract(\nu'(x)) \leq fract(\nu'(y)))$.
- $\forall x \in C \mid (\nu(x) \neq \infty)$, $(fract(\nu(x)) = 0 \Leftrightarrow fract(\nu'(x)) = 0)$.

Où $\lfloor \nu(x) \rfloor$ *et* $fract(x)$ *représentent respectivement la partie entière et la partie fractionnaire du nombre* x.

Une région d'horloge d'un *AEST A* est une classe d'équivalence d'interprétations d'horloges induite par ~.

[v] désigne la région d'horloges dont ν fait partie. Les régions d'horloges de l'AEST de la figure 4.2 sont représentées dans la figure 4.3. Nous distinguons trois types de régions dans cette figure : Les points corners {1, 2, 3, 4}, les segments ouverts {5, 6, 7, 8, 9, 10, 11, 12, 13} et les régions ouvertes {14, 15, 16, 17, 18}.

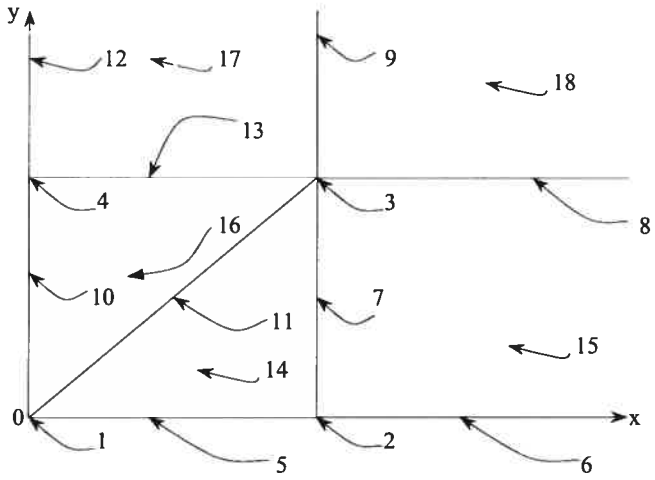

Figure 4.3 : Exemple de régions d'horloges

En se basant sur la définition 4.2, nous représentons le modèle sémantique d'un AEST A par un automate fini, dit *automate de régions* ou *graphe de régions.*

Définition 4.3 : *Graphe de Régions*

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

- $\Sigma = I \cup O \cup \mathbb{R}^{>0}$.
- $S = \{\langle l, [v]\rangle \mid l \in L \wedge v \in V(C)\}$.
- $s_0 = \langle l_0, [v_0]\rangle$, 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 la base de la vérification et du test des systèmes temps réel car il représente toutes les traces possibles du système décrit par l'*AEST*.

La figure 4.4 montre le graphe de régions de l'*AEST* de la figure 4.2. Chaque état de ce graphe est formé d'un emplacement de l'automate d'origine de la figure 4.2 et d'une région d'horloges de la figure 4.3. Certaines régions d'horloges sont inaccessibles et ne figurent pas donc sur le graphe. Quand aux transitions du graphe, elles sont formées des transitions explicites sur les entrées/sorties et des transitions de délai (étiquetées par le symbole *d* sur la figure).

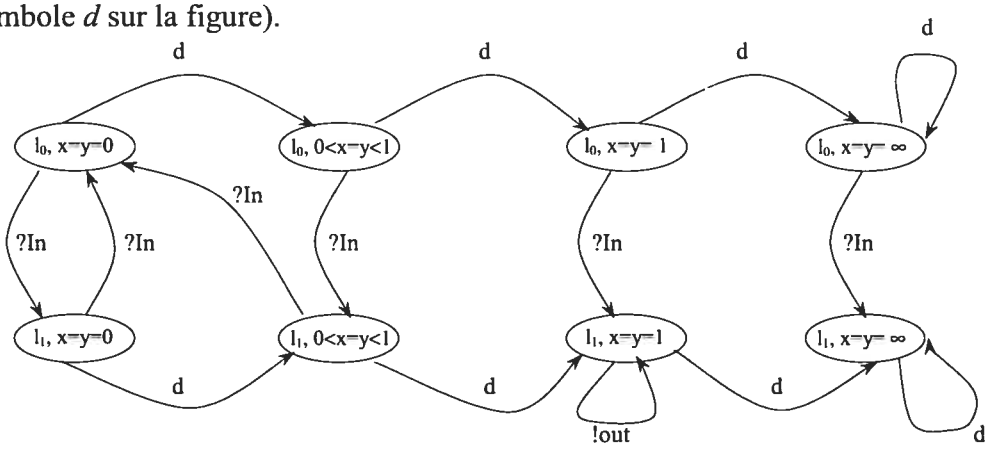

Figure 4.4 : Le graphe de régions de l'AEST de la Figure 4.2

L'approche de génération de tests consiste en plusieurs étapes. Dans la première étape, le graphe de région de la spécification est échantillonné 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 automate de grilles. Ce dernier présente explicitement l'écoulement de temps et il permet de tester l'IST par rapport à une bissimulation temporelle modulo k. Dans la deuxième étape, l'automate de grilles est transformée en une machine à états finis temporisée non déterministe observable (MEFTN). Ceci 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 des ensembles de caractérisation qui permettent 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, la MEFTN minimale est utilisée pour lui appliquer une version adaptée de la méthode Wp généralisée parallèlement à l'implantation.

4.6 Conclusion

Dans ce chapitre, nous avons passé en revue les méthodes de génération des cas de test à partir des modèles temporels. Pour chacune des méthodes présentées, nous avons expliqué les avantages, les inconvénients et les conditions d'application.

Comparativement aux autres méthodes de génération des cas de test temporisés [MAND 95][LIUG 93][CLAR 97][SPRI 97][ENDE 97], la dernière méthode (Wp temporisée) a l'avantage d'être générale et pratique avec une couverture complète de fautes sous le modèle et la granularité d'échantillonnage précédemment définies. Elle est basée sur les AESTs qui sont un modèle riche pour la spécification des systèmes temps réel. De plus, elle génère un nombre restreint de cas de test. Quant à la couverture de fautes, les cas de test générés sont capables de détecter toutes les fautes temporelles [ENNO 01][ENNO 02]. Dans le chapitre suivant, nous présenterons notre approche de diagnostic des systèmes temps réel modélisés par les AESTs. La méthode Wp temporisée y sera utilisée pour la détection des fautes à localiser et à corriger dans la suite du processus de diagnostic.

Chapitre 5

Méthode de diagnostic temps réel[1]

Dans ce chapitre, nous proposons une méthode de diagnostic [ELGH 02] pour le cas où une faute de sortie ou de transfert peut être présente dans un système temps réel modélisé par un AEST. La méthode Wp temporisée [ENNO 02] [SPRI 01] est utilisée pour la génération des cas de test. Le processus de diagnostic commence par une comparaison des sorties attendues à celles observées suite à l'exécution de ces cas de test. Si des symptômes sont détectés, nous générons un ensemble minimal de transitions suspectes dans la machine à états finis temporisée non déterministe (MEFTN), qui expliquent toutes les anomalies observées sur le comportement de l'implantation sous test. Chacune de ces transitions suspectes est ensuite identifiée dans l'automate de grilles puis dans l'AEST correspondant à la MEFTN. Notre méthode analyse la trace temporisée de l'exécution d'une suite de test pour déduire ensuite des hypothèses qui expliqueraient les erreurs détectées. Des cas de test additionnels sont parfois nécessaires pour la localisation de la faute.

Ce chapitre est organisé comme suit. La section 5.1 présente les principales étapes de la méthode Wp temporisée. Dans la section 5.2 nous détaillons notre méthode de diagnostic et nous illustrons ses différentes étapes à travers un exemple. Des conclusions seront formulées dans la section 5.3.

5.1 La méthode Wp temporisée : exemple illustratif

Dans cette section, nous présentons brièvement la méthode Wp temporisée. En particulier, nous introduisons le modèle de fautes ainsi que les différentes étapes de la méthode.

[1] Les résultats de ce chapitre on été publiés dans [ELGH 02]

5.1.1 Modèle de fautes

Le modèle de fautes pour les systèmes temps réel modélisés par des AESTs est formé de trois catégories de fautes : les fautes de transfert, de sortie et les fautes temporelles. 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. Par contre, elle est dite avoir une faute de sortie si elle ne répond pas par la sortie attendue dans l'un de ses états. 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 alors qu'elle ne doit pas l'être, soit à la restriction d'une contrainte d'une transition ou soit à l'élargissement d'une contrainte d'une transition.

Les figures ci-dessous présentent une spécification sous forme d'AEST et son implantation fautive. L'automate de la figure 5.1 a deux emplacements (l'emplacement initial l_0 et l'emplacement l_1) et deux transitions entre ces emplacements. Par exemple, la transition de l_0 vers l_1 est tirable sur la sortie *!p* lorsque la valeur de l'horloge x est comprise strictement entre 0 et 1. Cette transition remet à zéro l'horloge et ramène la machine à l'emplacement l_1. L'automate de la figure 5.2 correspond à l'implantation fautive de cette spécification. L'implantation présente une faute de transfert au niveau de la transition de l_1 vers l_0.

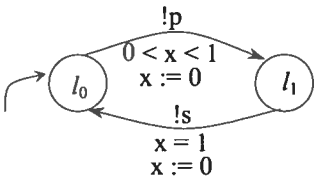

Figure 5.1 : Spécification sous forme d'AEST

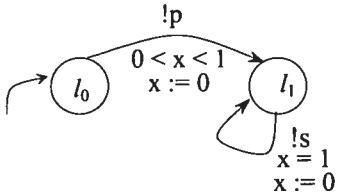

Figure 5.2 : Implantation fautive de l'AEST

5.1.2 Génération des cas de test à partir d'un AEST

Génération du graphe de régions

Comme nous l'avons déjà mentionné dans le chapitre précédent, la sémantique d'un AEST peut être donnée par un système de transitions étiquetées temporisé. Ce dernier est construit à partir des notions d'état et d'interprétations d'horloges et contient les transitions explicites et les transitions de délai. Cependant, ce système de transitions

étiquetées est infini à cause de l'infinité des transitions de délai (espace d'états infini). Pour réduire cet espace, la relation d'équivalence, définie dans la section 4.5 (page 42), permet de représenter le modèle sémantique d'AEST par un nombre fini d'états équivalents, sous forme de *graphe de régions*.

Les régions d'horloges de l'AEST de la figure 5.1 sont représentées dans la figure 5.3. Nous distinguons deux types de régions dans cette figure : les points corners $\{R_1, R_2\}$ et les segments ouverts $\{R_3, R_4\}$.

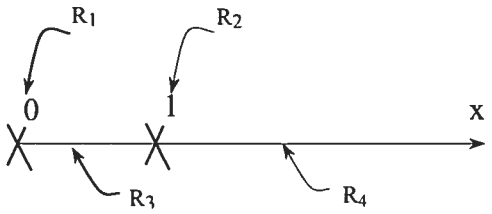

Figure 5.3 : Les régions d'horloges pour l'automate de la figure 5.1

La figure 5.4 présente le graphe de régions de la spécification (figure 5.1). Chaque état de ce graphe est formé d'un emplacement de l'automate d'origine (l_0 ou l_1) et d'une région d'horloges de la figure 5.3.

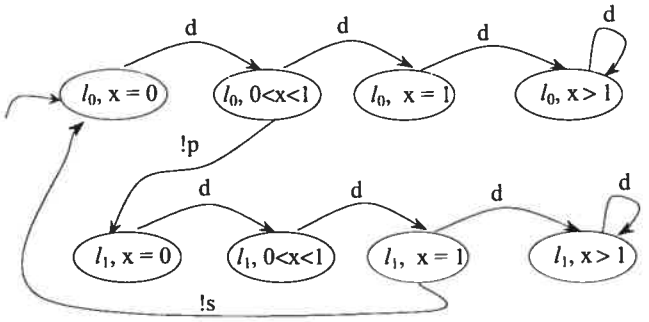

Figure 5.4 : Le graphe de régions de l'AEST de la figure 5.1

Échantillonnage du graphe de régions

Le but de l'échantillonnage est de construire un sous automate facilement testable, dit *automate de grille*. Ce dernier permet de représenter chaque région d'horloges par un ensemble fini d'interprétations d'horloges, appelées *représentants* de la région en question. Dans [ENNO 01], l'auteur démontre qu'une granularité de $1/(n+2)$ si $n \geq 2$, ou ½ sinon, est suffisante pour représenter toutes les régions d'horloges accessibles par des

transitions de délai d'un AEST à *n* horloges. Ainsi, la construction de l'automate de grille donne lieu à des actions de délai de valeur 1/(n+2) si n≥2, ou ½ sinon.

La granularité d'échantillonnage pour l'exemple précédent est de valeur ½ (puisque le nombre d'horloges est inférieur strictement à 2) et les représentants des différentes régions d'horloges sont alors {(0), (½), (1), (∞)}. Le symbole '∞' représente toutes les valeurs de l'horloge *x* supérieures strictement à 1. Nous obtenons ainsi l'automate de grille de la figure 5.5.

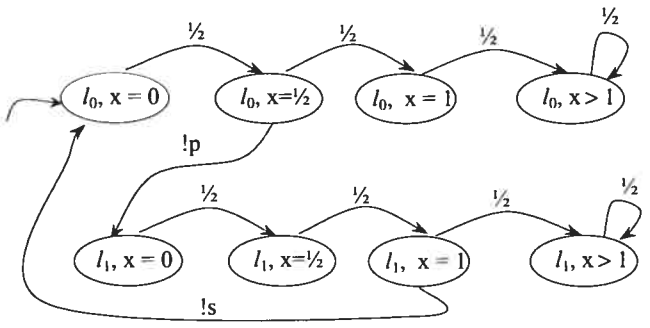

Figure 5.5 : Automate de grille

Transformation de l'automate de grille en une MEF

L'automate de grille est transformé en une machine à états finis temporisée non déterministe (*MEFTN*), où l'action correspondant à l'écoulement du temps est considérée comme une entrée. Cette MEFTN est ensuite minimisée en utilisant une technique légèrement différente de la minimisation classique des MEFs. La figure 5.6 présente la MEFTN minimale correspondant à l'automate de grille de la figure 5.5.

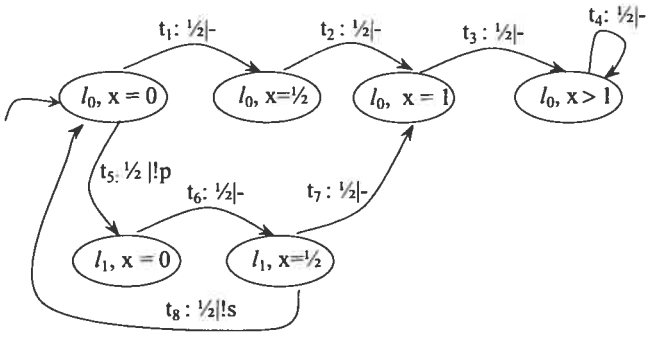

Figure 5.6 : La MEFTN minimale correspondant à l'automate de la figure 5.5

Génération de cas de test par la méthode Wp temporisée

Les transformations que nous avons effectuées sur l'AEST de la figure 5.1 introduisent d'abord l'action de délai ½ dans l'automate de grille et produisent une MEFTN minimale observable et complètement spécifiée (figure 5.6). Ainsi, sous l'hypothèse d'équité[2], la méthode Wp généralisée [LUOG 94] est adaptée puis appliquée pour la génération de la suite de test (TS) suivante :

$$\mathbf{TS = \{\ r.½,\ r.½.½,\ r.½.½.½,\ r.½.½.½.½,\ r.½.½.½.½.½\}\quad (*)}$$

Nous supposons qu'il existe une transition t_r permettant, à tout moment, de ramener la MEFTN minimale à l'état initial. Nous utilisons le symbole 'r' pour noter l'entrée de cette transition et le symbole '-' pour noter sa sortie.

Maintenant que la méthode Wp temporisée a été introduite, nous présentons notre méthode de diagnostic. Nous utilisons l'exemple des AEST des figures 5.1 et 5.2 et la suite de test précédemment générée (*) pour illustrer notre approche.

5.2. Méthode de diagnostic

Nous considérons que l'implantation sous test (IST) peut avoir une faute de sortie ou de transfert dans une et une seule de ses transitions. Notre méthode de diagnostic [ELGH 02] est constituée de 11 étapes que nous allons détailler dans cette section. Dans un souci didactique, chacune de ces étapes est illustrée à travers l'exemple de la section 5.1. Des algorithmes de transformation seront également proposés.

Étape 1 : Génération des sorties attendues

La suite de test (TS), obtenue à partir de la méthode W_p temporisée, est composée d'un nombre de cas de test qui sont des séquences de symboles d'entrée. Nous écrivons TS = (tc_1, tc_2, …, tc_p) où chaque tc_i est un cas de test. Comme le test porte sur une machine *non déterministe*, un *ensemble* de séquences de sorties est attendu pour chacun des cas de test tc_i de la suite TS. Cet ensemble couvre tous les chemins (séquences de transitions) qui peuvent être exécutés dans la spécification quand la séquence des symboles d'entrées

[2] L'hypothèse d'équité signifie que l'application d'un cas de test un nombre fini de fois permet de parcourir tous les chemins d'exécution possibles de l'implantation.

correspondante est appliquée. Ainsi, si un cas de test tc_i est formé de m_i entrées $I_{i,1}.I_{i,2}$ ….$I_{i,mi}$, l'ensemble des séquences de sorties attendues qui lui correspond est exprimé par : $O_i = \{O^1_i,…., O^q_i\}$, avec $O^k_i = \{O^k_{i,1}\ O^k_{i,2} \ldots O^k_{i,mi}\}$ sachant que la sortie $O^k_{i,j}$ (k = 1, …,q) se produit après l'entrée $I_{i,j}$.

Étape 2 : Génération des sorties observées

À cause de son indéterminisme, l'implantation peut générer, pour chaque cas de test tc_i, différentes séquences de sorties. L'ensemble de ces séquences est écrit comme suit :
$\hat{O}_i = \{\hat{O}^1_i,…., \hat{O}^\rho_i\}$, avec $\hat{O}^k_i = \{\hat{O}^k_{i,1}\ \hat{O}^k_{i,2} \ldots \hat{O}^k_{i,mi}\}$ et k = 1,....,ρ.

Exemple :

Les séquences de sorties observées et celles attendues pour l'exemple de la section 5.1 sont données dans le tableau 5.1 (page 52).

Étape 3 : Génération des symptômes

Dans cette étape, nous comparons l'ensemble des séquences de sorties *observées* $\hat{O}_i$ à l'ensemble des séquences de sorties *attendues* O_i, pour chaque cas de test tc_i, et nous identifions tous les symptômes. Toute différence ($\hat{O}_i \neq O_i$) représente un *symptôme*, noté par $Symp_i = (\hat{O}_i \neq O_i)$.

Une transition de la MEFTN de la spécification où un symptôme est observé, est appelée *transition symptôme*. La sortie observée $\hat{O}^k_{i,j}$ correspondant à la transition symptôme est appelée la *sortie symptôme*.

Si la même sortie symptôme $\hat{O}^k_{i,j}$ est observée à chaque fois à la place de la sortie attendue $O^k_{i,j}$ et elle est commune à toutes les transitions symptômes pour les différents cas de test, la sortie $\hat{O}^k_{i,j}$ est appelée l'*unique sortie symptôme* (*uss*) et l'ensemble des transitions symptômes correspondant est appelé *l'Ensemble des transitions symptômes uniques* (*Etsu*). Sinon cet ensemble *Etsu* est vide.

Si lors de cette étape l'Etsu est vide, nous passons à l'étape 5, sinon nous passons à l'étape 4.

Étape 4 : Identification de la transition fautive pour la faute de la sortie ssu

Pour chaque transition de l'Etsu, nous déterminons la transition suspecte correspondante dans l'AEST. Si l'ensemble résultant est un singleton $\{t_k\}$, alors t_k est une transition suspecte pour une faute de la sortie ssu. Si sous l'hypothèse que cette dernière est la sortie de la transition t_k dans l'AEST de la spécification, les sorties déjà observées et les sorties nouvellement attendues sont égales pour toutes les transitions dans tous les cas de test ; t_k est la transition fautive avec une faute de la sortie ssu et nous terminons la procédure de diagnostic.

Si l'ensemble résultant contient plusieurs transitions ou bien sous cette hypothèse les ensembles des sorties déjà observées et celles nouvellement attendues ne sont pas égales pour un certain cas de test, l'implantation contient éventuellement une faute de transfert simple et nous cherchons la transition suspecte en passant à l'étape 5.

Exemple :

Les différences entre les sorties observées et celles attendues sont détectées pour les cas de test tc_4 et tc_5 (tableau 5.1). Ainsi, les symptômes sont $\mathbf{Symp_4 = (\hat{O}_4 \neq O_4)}$ et $\mathbf{Symp_5 = (\hat{O}_5 \neq O_5)}$. l'ensemble Etsu est vide et nous passons à l'étape 5.

Étape 5 : Génération des ensembles de conflits

Pour chaque symptôme $(\hat{O}_i \neq O_i)$ détecté lors de l'application du cas de test tc_i à l'implantation, nous déterminons son ensemble de conflits, noté par $Conf_i$. Un ensemble de conflits est formé des transitions exécutées qui ont pu participer à la génération du symptôme. Cependant, quelques-unes de ces transitions sont réellement le résultat de la transition fautive dans l'AEST correspondant à l'implantation.

tc_1 : séquence d'entrées	r.½
Séquences de transitions attendues Séquences de sorties attendues Séquences de sorties observées	{t_r t_1} {t_r t_5} {- -} {- !p} {- -} {- !p}
tc_2 : séquence d'entrées	r.½.½
Séquences de transitions attendues Séquences de sorties attendues Séquences de sorties observées	{t_r t_1 t_2} {t_r t_5 t_6} {- - -} {- !p -} {- - -} {- !p -}
tc_3 : séquence d'entrées	r.½.½.½
Séquences de transitions attendues Séquences de sorties attendues Séquences de sorties observées	{t_r t_1 t_2 t_3} {t_r t_5 t_6 t_7} {t_r t_5 t_6 t_8} {- - - -} {- !p - -} {- !p - !s} {- - - -} {- !p - -} {- !p - !s}
tc_4 : séquence d'entrées	r.½.½.½.½
Séquences de transitions attendues Séquences de sorties attendues Séquences de sorties observées	{t_r t_1 t_2 t_3 t_4} {t_r t_5 t_6 t_7 t_3} {t_r t_5 t_6 t_8 t_1} {t_r t_5 t_6 t_8 t_5} {- - - - -} {- !p - - -} {- !p - !s -} **{- !p - !s !p}** {- - - - -} {- !p - - -} {- !p - !s -}
tc_5 : séquence d'entrées	r.½.½.½.½.½
Séquences de transitions attendues Séquences de sorties attendues Séquences de sorties observées	{t_r t_1 t_2 t_3 t_4 t_4} {t_r t_5 t_6 t_7 t_3 t_4} {t_r t_5 t_6 t_8 t_1 t_2} {t_r t_5 t_6 t_8 t_5 t_6} {- - - - - -} {- !p - - - -} {- !p - !s - -} **{- !p - !s !p -}** {- - - - - -} {- !p - - - -} {- !p - !s - -} **{- !p - !s - *!s*}**

Tableau 5.1 : Séquences de sorties observées et attendues

Dépendamment des fautes qui peuvent parvenir dans une IST sous forme d'un AEST, différentes situations peuvent se présenter. Par exemple, si nous appliquons une séquence d'entrées, générée par la méthode W_p temporisée, N fois à une implantation sous forme d'AEST qui contient une faute de sortie ou de transfert dans l'une de ses transitions, il est possible de manquer une ou plusieurs séquences de sorties attendues de l'implantation. Des nouvelles séquences de sorties peuvent apparaître.

- Si toutes ces nouvelles séquences de sorties sont identiques aux autres séquences attendues selon la spécification et observées dans l'implantation, l'ensemble $\hat{O}_i$ sera inclu dans O_i. Dans ce cas l'ensemble de conflits sera constitué de quelques-unes des transitions qui sont supposées générer les séquences de sorties manquantes.

- Un deuxième cas est celui où quelque-unes des nouvelles séquences de sorties sont identiques à celles attendues et les autres sont différentes de toutes les séquences de sorties attendues. Dans ce cas, ni l'ensemble $\hat{O}_i$ n'est inclus dans O_i, ni l'ensemble O_i n'est inclus dans $\hat{O}_i$. L'ensemble de conflits sera constitué de quelque-unes des transitions qui sont supposées générer les séquences de sorties manquantes, mais elles ont généré les nouvelles séquences à la place.

- La troisième et la dernière situation est le cas où, à cause des fautes de transfert, toutes les séquences de sorties attendues et des nouvelles séquences sont observées dans l'implantation. Dans ce cas, l'ensemble O_i est inclus dans $\hat{O}_i$ et l'ensemble de conflits correspondant est constitué de quelque-unes des transitions participant à la génération des extra-séquences de sorties observées.

Nous présentons une description 'formelle' de la procédure permettant de générer les ensembles de conflits :

Si $\hat{O}_i \subset O_i$ alors
$Plus_i = O_i - \hat{O}_i$
$Checkset_i = \hat{O}_i$
ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, O_i, flag)

Sinon si $O_i \subset \hat{O}_i$ alors
$Plus_i = \hat{O}_i - O_i$
$Checkset_i = O_i$
ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, O_i, flag)

Sinon
$Plus_i = O_i - \hat{O}_i$
$Checkset_i = \hat{O}_i - O_i$
ComputeConf($Conf_i$, $Plus_i$, $Checkset_i$, O_i, flag)

Procedure **ComputeConf**(Conf, Plus, Checkset, O, flag)

$Conf := \bigcap_{\sigma \in Plus} Conf_{\sigma'}$ ou l'ensemble $Conf_\sigma$ est calculé comme suit :
Chercher la séquence $\sigma' \in$ Checkset qui a le plus grand préfixe commun avec σ :

Si $\sigma = x_1x_2x_3.....x_mx_{m+1}..x_n$, $\sigma' = x_1x_2x_3.....x_mx'_{m+1}..x'_n$ si
pour toute séquence $\sigma'' = x_1x_2x_3.....x_{m'}x''_{m'+1}..x''_n$ $m' \leq m$

$Conf_\sigma = \{t_i \mid t_i$ est la transition de la spécification déterminée par l'entrée I_i, la sortie x_i et sa position i dans la séquence σ tel que $2 \leq i \leq m \} \cup \{t_{m+1} \mid t_{m+1}$ est déterminée par l'entrée i_{m+1}, la sortie y_{m+1} et sa position dans la séquence correspondante, telle que $y_{m+1} = x_{m+1}$ si $\sigma \in O$, sinon, $y_{m+1} = x'_{m+1}$ $\}$

Chaque appel à la procédure 'ComputeConf' génère l'ensemble de conflits minimal '*Conf*' correspondant aux symptômes considérés. Cet ensemble de conflits contient toujours l'ensemble des transitions, de la MEFTN, résultats de la transition réellement fautive de l'implantation.

En appliquant cette méthode, les ensembles de conflits correspondants aux symptômes $Symp_4$ et $Symp_5$ sont les suivants : $\mathbf{Conf_4 = \{t_5, t_6, t_8\}}$, $\mathbf{Conf_5 = \{t_5, t_6, t_8\}}$.

Étape 6 : Génération des transitions suspectes dans la MEFTN

Une transition suspecte d'être fautive dans la MEFTN doit appartenir à chacun des ensembles de conflits générés dans l'étape 5, et doit être consistante avec toutes les observations. *L'intersection* de tous les ensembles de conflits permet de générer un ensemble *I* de transitions initialement suspectes. Chacune de ces transitions est perçue comme un candidat plausible qui est susceptible d'expliquer tous les symptômes. Il importe de noter que l'unique transition fautive de l'implantation sous forme d'AEST conduit à une ou plusieurs transitions fautives au niveau de la MEFTN correspondante. Toutes ces transitions fautives appartiennent à l'ensemble *I*.

Une transition suspecte t_k qui est de la forme 1/(n+2)|- ou ½|- représente un écoulement de temps tout en restant dans un même emplacement. En supposant que ces transitions ne causent pas de faute de transfert, nous pouvons les éliminer de l'ensemble des transitions suspectes. L'ensemble obtenu est noté par I_{Conf}.

Comme nous l'avons précédemment mentionné, l'unique transition fautive de l'implantation conduit à une ou plusieurs transitions fautives au niveau de la MEFTN correspondante. Toutes ces transitions appartiennent à I_{Conf}.

Exemple :

$I = Conf_4 \cap Conf_5 = \{t_5, t_6, t_8\}$. La transition t_6 est une transition de la forme ½|-. L'ensemble I_{Conf} des transitions suspectes est formé de $\mathbf{t_5}$ **et** $\mathbf{t_8}$.

Étape 7 : Identification des transitions suspectes dans l'Automate de Grille

Pour chaque transition t_k de l'ensemble des transitions suspectes dans la MEFTN, nous localisons la ou les deux transitions équivalentes à t_k dans l'automate de grille.

La figure 5.7 montre les différents schémas de transformation qui peuvent servir à l'identification des transitions suspectes dans l'automate de grille (AG). Dans cette figure le symbole g représente la granularité de l'échantillonnage (½ ou 1/(n+2)).

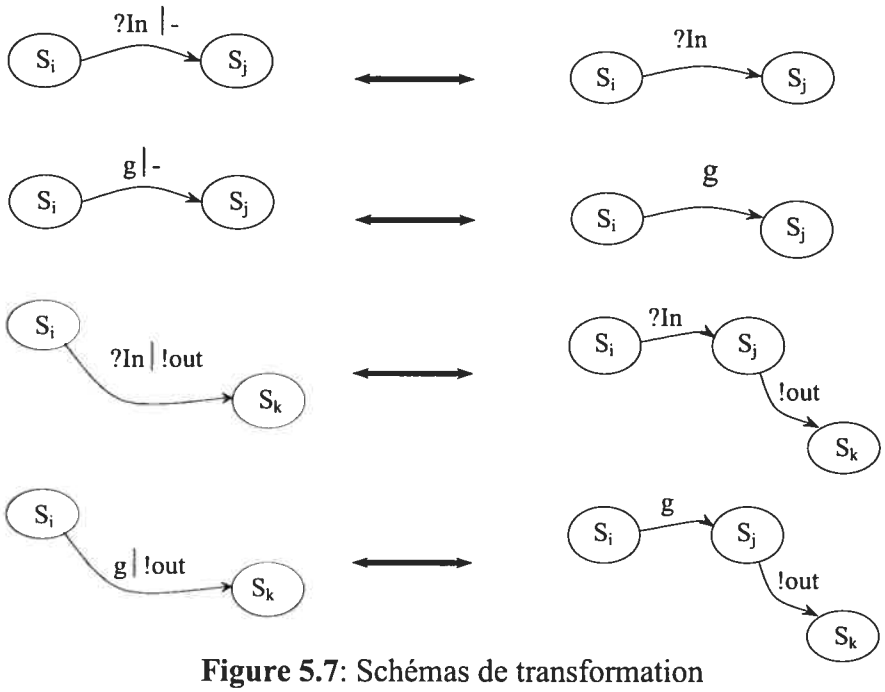

Figure 5.7: Schémas de transformation

L'algorithme de ces transformations est décrit dans la figure 5.8. Il a comme entrée un AEST de la spécification ainsi que l'ensemble I_{Conf} et produit comme résultat l'ensemble $susp_{Gr}$ des transitions suspectes au niveau de l'automate de grille.

Pour construire l'ensemble $susp_{Gr}$, notre algorithme procède comme suit. Dans la première étape, nous initialisons les ensembles à utiliser. Dans la deuxième étape, nous choisissons une transition de la forme $((l,v), granul|!out, (l',v'))$ non encore traitée, de l'ensemble I_{Conf}, et nous ajoutons la transition $((l,v+granul), !out, (l',v'))$ à l'ensemble $susp_{Gr}$ pour toute transition $(l, l', !out, R,G)$ de l'AEST telle que $(v+granul)$ satisfait G et $v' = [R:= 0](v+granul)$. Dans la troisième étape, nous choisissons une transition de la forme $((l,v), ?in|!out, (l'',v''))$ non encore traitée et nous ajoutons à l'ensemble $susp_{Gr}$ la transition $((l,v), ?in, (l',[R:= 0]v))$, pour toute transition $(l, l', ?in, R,G)$ de l'AEST telle que v satisfait G. Puis, nous ajoutons la transition $((l',[R:= 0]v), !out, (l'', [R':= 0]([R:= 0]v)))$ pour toute transition $(l', l'', !out, R', G')$ de l'AEST telle que $[R:= 0]v$ vérifie G'

Exemple :

La figure 5.9 montre les deux transitions équivalentes à t_5 et t_8 dans l'automate de grille de la figure 5.5.

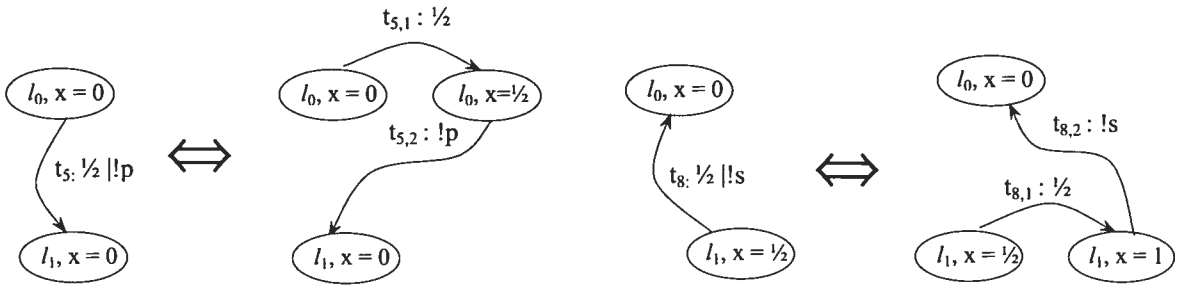

Figure 5.8 : Les deux transitions équivalentes à t_5 et t_8 dans l'automate de grilles

Étape 8 : Identification des transitions suspectes dans l'AEST

Dans cette étape, nous déterminons l'ensemble des transitions suspectes au niveau de l'AEST, que nous noterons *Susp*, correspondant aux transitions suspectes de l'ensemble $susp_{Gr}$. Pour construire l'ensemble Susp des transitions suspectes au niveau de l'AEST l'algorithme procède comme suit. Dans la première étape, nous initialisons les ensembles à utiliser. Dans la deuxième étape, nous choisissons une transition $((l,\nu), \{!\ ?\}a\ , (l',\nu'))$ non encore traitée, de l'ensemble des transitions suspectes au niveau de l'automate de grilles, et nous ajoutons à l'ensemble Susp chaque transition $(l, l', \{!\ ?\}a, R, G)$ de L'AEST telle que ν satisfait G et $\nu'=[R:=0]\nu$.

Entrées : - L'ensemble, I_{Conf}, des transitions suspectes au niveau de la MEFTN

- L'AEST de la spécification $A = (I, O, L, l^0, C, T)$

Sorties : - Un ensemble, $susp_{Gr}$, de transitions suspectes au niveau l'Automate de Grilles

Étape 0 : Initialisation des ensembles à utiliser

$susp_{Gr} = \emptyset$,

$H_1 = \emptyset$ (1er sous-ensemble de transitions traitées)

$H_2 = \emptyset$ (2ème sous-ensemble de transitions traitées)

$susp' = \emptyset$.

Étape1: Construction du 1er sous-ensemble de $susp_{Gr}$

Tant Qu'il existe une transition $T = ((l,\nu), granul \mid !out, (l',\nu')) \in I_{Conf} \backslash H_1$ Faire

$H_1 = H_1 \cup T$

Si $out \neq$ `-`

Pour chaque transition $(l, l', !out, R, G) \in A$ Faire

Si $(\nu + granul) \models G \;\&\&\; \nu' = [R:=0](\nu + granul)$

$susp_{Gr} = susp_{Gr} \cup ((l,\nu + granul), !out, (l', \nu'))$

FinSi

FinPour

FinSi

FinTandQue

Étape2: Construction du 2ème sous-ensemble de $susp_{Gr}$

$susp' = I_{Conf} \backslash H_1$

Tant Qu'il existe une transition $T = ((l,\nu), ?in|!out, (l'',\nu'')) \in susp' \backslash H_2$ Faire

$H_2 = H_2 \cup T$

Pour chaque transition $(l, l', ?in, R, G) \in A$ Faire

Si $\nu \models G$

$susp_{Gr} = susp_{Gr} \cup ((l,\nu), ?in, (l',[R:=0]\nu))$

Si $out \neq$ `-`

Pour chaque transition $(l', l'', !out, R', G') \in A$ Faire

Si $[R:=0]\nu \models G' \;\&\&\; \nu'' = [R':=0]([R:=0]\nu)$

$susp_{Gr} = susp_{Gr} \cup ((l',[R:=0]\nu), !out, (l'', [R':=0]([R:=0]\nu)))$

FinSi

FinPour

FinSi

FinSi

FinPour

FinTantQue

Figure 5.9 : Algorithme de correspondance de transitions MEFTN/AG

Entrées : - Un ensemble, $susp_{Gr}$, de transitions suspectes au niveau de l'automate de grilles.

- L'AEST de la spécification $A=(I, O, L, l^0, C, T)$.

Sorties : - Un ensemble, Susp, de transitions suspectes au niveau l'AEST de l'implantation

Étape 0 : Initialisation des ensembles à utiliser

Susp = ∅

H_1 = ∅ (Transitions Traitées)

Étape1: Construction de l'ensemble Susp

Tant que $susp_{Gr} \backslash H_1 \neq \emptyset$ Faire

Choisir une transition $T= ((l,v), \{!\ ?\}a\ , (l',v')) \in susp_{Gr} \backslash H_1$

$H_1 = H_1 \cup T$

Pour chaque transition $(l, l', \{!\ ?\}a, R,G) \in A$ Faire

Si $v \models G$ && $v' = [R:=0]v$

Susp = Susp ∪ $(l, l', \{!\ ?\}a, R,G)$

FinSi

FinPour

Fin Tant Que

Figure 5.10 : Algorithme de correspondance de transitions AG/AEST

$t_{5,2}$ correspond à la transition de l_0 vers l_1 (soit $t^5_{l0,l1}$) dans l'AEST de la figure 5.1 quand la valeur de l'horloge x est égale à ½. t_5 est suspecte d'une faute de transfert signifie donc que $t^5_{l0,l1}$ est suspecte d'une faute de transfert. De même, $t_{8,2}$ correspond dans la figure 5.1 à la transition de l_1 vers l_0 (soit $t^8_{l1,l0}$) dans l'AEST lorsque x = ½, et t_8 est suspecte d'une faute de transfert signifie que $t^8_{l1,l0}$ est suspecte d'une faute de transfert. Nous obtenons ainsi l'ensemble suivant des transitions suspectes au niveau de l'AEST : **Susp** = $\{t^5_{l0,l1}, t^8_{l1,l0}\}$.

Étape 9 : Détermination des emplacements de transfert fautif

Notons que dans le cas d'AEST complexes, les étapes précédentes permettent généralement une réduction considérable des transitions suspectes.

Dans cette étape, pour chaque transition $t^k_{l,l'}$ dans Susp, nous déterminons l'ensemble des emplacements auxquels $t^k_{l,l'}$ pourrait transférer à l'exception de l'emplacement qui suit $t^k_{l,l'}$ dans la spécification.

Pour chaque emplacement *e* considéré, *e* sera inclus dans un ensemble dit *ensemble d'emplacements de transfert fautif* et noté par *Empl*($t^k_{l,l'}$), si :

Sous l'hypothèse que *e* est l'emplacement qui suit $t^k_{l,l'}$ dans la spécification et en considérant la MEFTN correspondante, les sorties nouvellement attendues et les sorties observées auparavant sont égales pour toutes les transitions dans tous les cas de test.

Exemple :

Dans notre cas, le seul emplacement où $t^5_{l0,l1}$ pourrait transférer est l'emplacement l_0. Nous supposons que $t^5_{l0,l1}$ transfère à l'emplacement l_0 et nous obtenons l'AEST de la figure 5.11 auquel correspond la MEFTN de la figure 5.12.

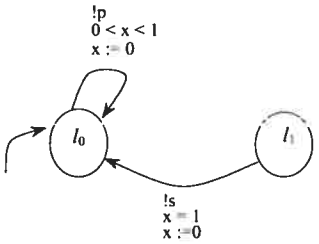

Figure 5.11 : L'AEST obtenu sous l'hypothèse que $t^5_{l0,l1}$ transfère à l'emplacement l_0 au lieu de l_1

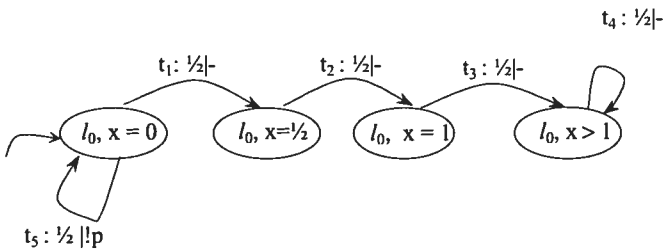

Figure 5.12 : La MEFTN correspondant à l'AEST de la figure 5.11

Les sorties observées auparavant (tableau 5.1) et les sorties nouvellement attendues ne sont pas égales pour tous les cas de test. L'emplacement l_0 ne sera donc pas inclus dans l'ensemble Empl($t^5_{l0,l1}$) et nous obtenons **Empl($t^5_{l0,l1}$) = Ø**.
En appliquant l'étape 9 également pour la transition $t^8_{l1,l0}$, nous obtenons **Empl($t^8_{l1,l0}$)** = $\{l_1\}$.

Étape 10 : Réduction du nombre de transitions suspectes et déduction d'hypothèses

Toute transition $t^k_{l,l'}$ dont l'ensemble des emplacements de transfert fautif Empl($t^k_{l,l'}$) est vide n'est plus suspecte, elle peut donc être enlevée de l'ensemble des transitions suspectes Susp construit à l'étape 8. Nous obtenons ainsi un ensemble de transitions suspectes réduit, noté par *RSusp*.
Pour chaque transition $t^k_{l,l'} \in$ RSusp et pour chaque emplacement $e \in$ Empl($t^k_{l,l'}$), l'hypothèse suivante est déduite :
la transition $t^k_{l,l'}$ transfère à e au lieu de l'emplacement qui suit $t^k_{l,l'}$ dans la spécification.
Notons que la transition réellement fautive sera dans l'ensemble Rsusp, puisque toutes les transitions de l'AEST de la spécification relatives à l'intersection des ensembles de conflits, y compris la transition réellement fautive, sont examinées pour toutes les fautes de transfert.

Étape 11 : Génération éventuelle des cas de test additionnels

Selon les résultats des étapes précédentes, deux cas peuvent se présenter :

Cas 1 : *tests additionnels requis*
Lorsque plusieurs éventualités persistent, d'autres tests supplémentaires sont nécessaires pour identifier la transition fautive et l'emplacement auquel elle transfère. Ce cas se manifeste dans l'une des deux situations suivantes :

- l'ensemble des transitions suspectes réduit contient plusieurs transitions ;
- l'ensemble des transitions suspectes réduit est un singleton (RSusp = $\{t^k_{l,l'}\}$) avec un ensemble d'emplacements de transfert fautif qui contient plus qu'un élément.

Pour chaque transition $t^k_{l,l'}$ de RSusp, les cas de test additionnels sélectionnés sont exécutés dans l'objectif de distinguer entre les différents emplacements contenus dans l'ensemble Empl($t^k_{l,l'}$). Ainsi, un ensemble de caractérisation W_k doit être calculé.

Cet ensemble concerne uniquement le sous ensemble Empl($t^k_{l,l'}$) et non tous les emplacements de l'AEST. W_k sera formé de séquences d'entrées telles que :

Si ces séquences d'entrées sont appliquées dans l'un des emplacements de Empl($t^k_{l,l'}$), les sorties produites seront différentes de celles obtenues si les mêmes séquences d'entrées sont appliquées dans un autre emplacement du même ensemble.

Pour chaque emplacement $l_{k,i}$ de l'ensemble Empl($t^k_{l,l'}$), et sous l'hypothèse que $l_{k,i}$ est l'emplacement suivant $t^k_{l,l'}$, nous considérons la MEFTN (déjà générée à l'étape 7) correspondante à la nouvelle spécification. La transition $t^k_{l,l'}$ transfère maintenant vers un état $S_{k,i} = (l_{k,i}, \nu)$ dans la nouvelle MEFTN. Chaque cas de test additionnel est une concaténation d'une séquence d'entrées, dites séquence de transfert, pouvant amener cette nouvelle machine depuis l'état $S_0 = (l_0, \nu_0)$ à l'état initial de $t^k_{l,l'}$, de l'entrée de $t^k_{l,l'}$ et d'une séquence de caractérisation de W_k.

La construction des tests additionnels est progressive. Quand la faute est localisée, le reste des tests additionnels n'est plus nécessaire.

Cas 2 : *aucun test additionnel n'est requis*

L'ensemble des transitions suspectes réduit est un singleton (RSusp = $\{t^k_{l,l'}\}$) avec un seul emplacement de transfert fautif l_k (Empl($t^k_{l,l'}$) = $\{l_k\}$). Dans ce cas, nous concluons que la transition $t^k_{l,l'}$ a une faute de transfert à l'emplacement l_k et aucun test additionnel n'est nécessaire.

Dans notre exemple, l'ensemble des transitions suspectes réduit est un singleton avec un seul emplacement de transfert fautif :

$$\text{RSusp} = \{t^8_{l1,l0}\} \text{ et Empl}(t^8_{l1,l0}) = \{l_1\}.$$

Ceci correspond au cas où aucun test additionnel n'est nécessaire, et nous arrivons à la conclusion suivante :

$t^8_{l1,l0}$ transfère à l'emplacement l_1 au lieu de l_0.

5.3 Conclusion

Nous avons proposé une méthode de diagnostic qui permet la localisation d'une faute de sortie ou de transfert dans un système temps réel modélisé par un AEST. La méthode Wp temporisée a été utilisée pour générer les cas de test. Suite à une comparaison des sorties observées avec les sorties attendues, un ensemble minimal de transitions suspectes a été généré. Ces transitions ont été dans un premier temps identifiées au niveau de MEFTN. Ensuite, elles ont été localisées dans l'automate de grille et dans l'AEST correspondant à cette MEFTN. La détermination des emplacements de transfert fautif a conduit à une réduction du nombre de transitions suspectes et à la déduction d'hypothèses. Des cas de test additionnels se sont avérés nécessaires dans certains cas pour garantir la localisation de la faute de transfert.

Chapitre 6

Présentation de l'outil de diagnostic

Dans ce chapitre, nous présentons l'outil de diagnostic des systèmes temps réel, que nous appelons 'RTDiag' (pour 'Real Time Diagnostic'). Cet outil analyse la trace d'exécution, et si un symptôme est détecté, il produit un ensemble minimal de transitions suspectes qui peuvent expliquer le comportement de l'implantation. Les différents symptômes sont ensuite combinés pour obtenir des hypothèses globales sur le comportement de l'implantation. 'RTDiag' permet aussi de valider chacune de ces hypothèses par la technique de mutation appliquée à l'ensemble de transitions suspectes déjà obtenu.

6.1 Fonctionnalités de l'outil

'RTDiag' implémente l'approche de diagnostic que nous avons développée dans le chapitre 5. Pour réduire l'espace des candidats possibles et localiser la faute, l'outil exécute les tâches suivantes:

- Lecture de la spécification en AEST à partir d'un fichier texte, qui doit respecter une grammaire spécifique, puis traduction de cette spécification en structures de données appropriées en utilisant la librairie des graphes GTL (pour 'Graph Template Library').

- Construction de l'Automate de Grille (AG) correspondant à la spécification après avoir calculé la granularité de l'échantillonnage.

- Transformation de l'AG en MEFTN minimale facilement testable comme c'est expliqué dans le chapitre précédent et cela en utilisant toujours la librairie GTL.

- Application de la suite de test sur la MEFTN résultante pour obtenir le comportement normal d'une implantation correcte sous forme de sorties (dites sorties attendues). Ces sorties sont comparées à celles extraites de la trace d'exécution. Si elles ne sont pas identiques, et pour chaque cas de test qui contient des différences, un ensemble de transitions symptômes est calculé selon la procédure présentée à l'étape 5 de l'algorithme de diagnostic (section 5.2).

- Les hypothèses issues des différents cas de test sont combinées pour avoir un seul ensemble de transitions suspectes, au niveau de la MEFTN, qui servira pour identifier les transitions suspectes dans l'automate de grilles puis dans l'AEST. Pour effectuer cette tâche, les deux algorithmes de transformation proposés à l'étape 7 de la section 5.2 sont appliqués.

- Pour discriminer entre les candidats, 'RTDiag' construit la machine mutante correspondant à chaque transition suspecte de l'AEST et compare encore une fois les sorties nouvellement attendues et les sorties observées. Dans le cas où plusieurs machines mutantes correspondent au comportement de l'implantation, des cas de test additionnels seront calculés et appliqués pour distinguer entre ces machines.

6.2 Architecture conceptuelle du système

Nous avons décomposé le système en plusieurs modules, chacun réalise une tâche bien particulière tout en restant ouvert à une communication avec les autres modules.
La figure 6.1 présente le premier niveau de décomposition de RTDiag. Ce dernier reçoit en entrée une spécification sous forme d'AEST, les cas de test correspondants ainsi que la suite de sorties observées après l'exécution de ces cas de test sur une spécification erronée et produit un diagnostic si des symptômes se présentent.

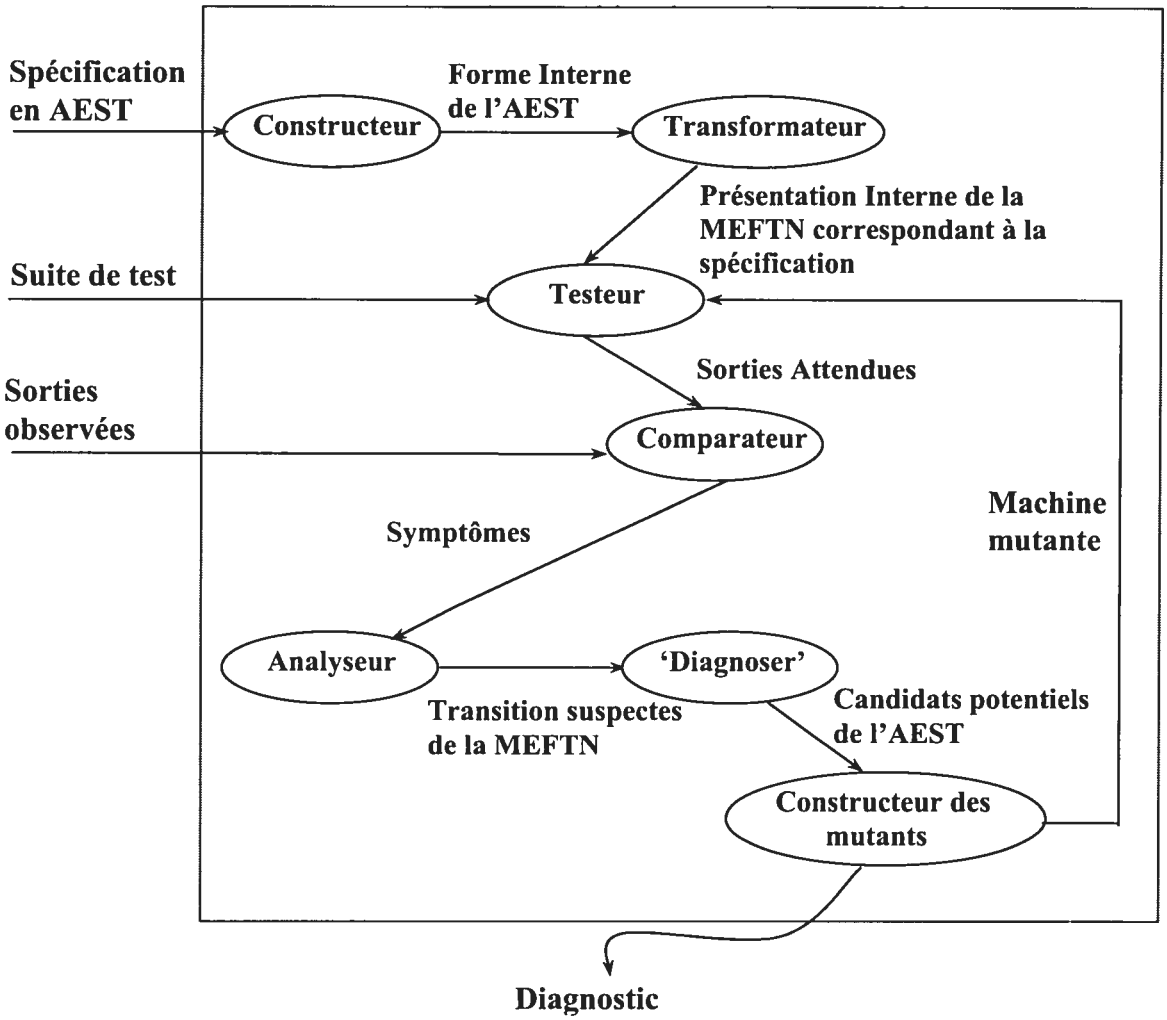

Figure 6.1 : Décomposition de 'RTDiag' en modules

Nous présentons dans ce qui suit le rôle de chacun des modules dans le schéma ci-dessus :

- **Constructeur** : Ce premier module reçoit la spécification du système sous forme d'un fichier texte qui doit respecter une syntaxe bien spécifique. Le rôle du constructeur est de transformer cette spécification en une forme interne, sans perte d'information. La forme interne est construite en utilisant la librairie GTL. Cette librairie est devenue un standard pour les concepteurs logiciels qui travaillent avec les graphes complexes et les algorithmes associés. GTL contient un ensemble de classes relatives à la structure de données d'un graphe (Nœud, Arc, ...etc) et les algorithmes les plus répondus pour leurs manipulations dans plusieurs domaines

d'applications comme l'informatique, les mathématiques discrètes, la chimie, etc. GTL est faite à base des STLs (pour 'Standard Template Lirary') qui font maintenant partie de la librairie standard de C++.

- **Transformateur** : Ce module permet de transformer la forme interne de la spécification en automate de grilles puis en Machine à États Finis Temporisé Non déterministe Minimale (MEFTNM) selon les algorithmes de transformation [ENNO 01] pour pouvoir lui appliquer les séquences de test générées par la méthode Wp temporisée. La MEFTNM résultante est conçue à l'aide des GTLs.

- **Testeur** : Les sorties observées sont directement extraites à partir de la trace d'exécution. Pour avoir les sorties attendues, ce module applique la séquence de test à la MEFTNM. Les cas de test sont appliqués l'un après l'autre et pour chaque entrée d'un cas de test, le testeur exécute les tâches suivantes :
  - recherche les transitions ayant l'entrée en question.
  - recherche parmi ces transitions celle qui sera exécutée en tenant compte des valeurs des variables d'état.
  - exécute la transition retenue, met la sortie correspondante dans la pile des sorties et passe à l'état suivant.

- **Comparateur** : Comme son nom l'indique, ce module compare les sorties observées aux sorties attendues, recueillies à la sortie du testeur, pour chaque cas de test. Si les deux suites de sorties sont identiques le comparateur passe aux séquences de sorties suivantes sinon, il génère un symptôme qui sera analysé par la suite.

- **Analyseur** : Ce module applique la procédure de l'étape 5 de notre algorithme de diagnostic pour générer un ensemble de conflits correspondant à chaque symptôme détecté. L'exécution de ce module génère un ensemble de transitions suspectes pour le symptôme en question. Ensuite tous les symptômes sont considérés pour générer un seul ensemble de transitions suspectes au niveau de la MEFTNM.

- **'Diagnoser'** : L'analyseur nous a permis jusqu'à maintenant d'identifier un ensemble de transitions suspectes dans la MEFTNM. Le rôle du 'diagnoser' est de

localiser, pour chacune de ces transitions, la ou les deux transitions suspectes équivalentes dans l'automate de grilles puis dans l'AEST de départ. Pour cela, nous appliquons respectivement les deux algorithmes que nous avons proposés à l'étape 7 du processus de diagnostic (chapitre 5).

- **Constructeur des mutants** : Ce module reçoit en entrée les transitions suspectes générées par le 'diagnoser'. Son rôle est de trouver les causes qui expliqueraient les symptômes détectés. À partir de l'AEST de la spécification et pour chacune des transitions suspectes, ce module génère une hypothèse sur l'emplacement où la transition pourrait transférer. Ensuite, il construit la machine mutante en AEST respectant cette hypothèse. Cette machine est passée au testeur pour lui appliquer la séquence de test de départ et puis au comparateur. Si les sorties déjà observées et les sorties nouvellement attendues sont égales pour tous les cas de test, la transition suspecte est maintenue ; sinon, elle est éliminée de l'ensemble des suspects. Dans le cas où plusieurs transitions sont maintenues, ce module génère des cas de test additionnels pour discriminer entre elles.

6.3 Implantation du système

Tout au long du développement de l'outil, nous avons considéré l'éventualité de son extension et/ou la réutilisation de certains de ses modules. L'application d'une autre méthode de diagnostic sur le même modèle de spécification en AEST sera un exemple d'extension. Pour minimiser l'effort de maintenance et de réutilisation, nous avons opté pour une conception orientée objet. La notation utilisée est l'UML pour (Unified Modeling Langage «langage de modélisation objet unifié»); c'est un support de communication performant, qui facilite la représentation et la compréhension de solutions orientées objet :

- La notation graphique permet d'exprimer visuellement une solution objet, ce qui facilite la comparaison et l'évaluation des solutions.

- L'aspect formel de sa notation limite les ambiguïtés et les incompréhensions.

- Son indépendance par rapport aux langages de programmation, aux domaines d'application et aux processus, en font un langage universel.

6.3.1 Éléments de modélisation

De point de vue statique, nous avons décomposé le système en trois vues appelées 'packages'. Ces deniers permettent de structurer le système en catégories selon des critères purement logiques. Ils servent aussi de "briques" de base pour la construction d'une architecture et représentent le bon niveau de granularité pour la réutilisation. Chaque package possède une interface à travers laquelle il peut interagir avec un ou plusieurs autres packages selon un certain type de relation.

La figure 6.2 représente les trois packages de notre système ainsi que leurs interactions : Le package 'GridAutomata-NTFSMView' par exemple utilise les services des éléments du package 'TIOAView'.

La modélisation UML de 'RTDiag' est faite avec 'Rational Rose' 2002

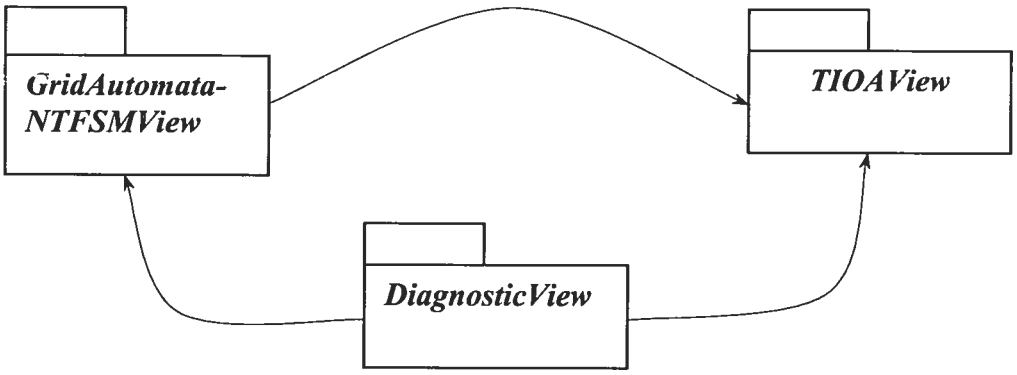

Figure 6.2: Les trois packages du système et leurs interactions

Chaque package identifie un ensemble d'éléments du domaine, ainsi que les relations et les interactions entre ces éléments selon un diagramme dit *diagramme de classes*. Ce dernier est le plus important dans une modélisation orientée objet. Il saisit la structure statique du système en identifiant un ensemble de classes, d'interfaces et de

collaborations, ainsi que leurs relations. Il constitue aussi le cadre dans lequel l'aspect dynamique s'insère.

TIOA-View : Cette vue représente le diagramme de classes qui permettent de charger en mémoire la structure de données d'un AEST. Une instance de la classe 'TIOA' hérite des propriétés de la classe 'graph' de la librairie GTL et elle est composée des instances des classes 'OutputAction', 'InputAction', 'map<edge, TIOATransition>' et 'Location' etc. Le lien entre la classe TIOA et ces dernieres dans le diagramme de la figure 6.3 exprime cette composition par un losange. Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés. L'association vers la classe 'Clock' exprime une relation unidirectionnelle.

Une instance de la classe map<edge,TIOATransition>, appartenant à la librairie STL, permet d'associer à chaque instance de la classe 'edge' de la librairie GTL, une instance de la classe 'TIOATransition' et ce pour un ou plusieurs couples de classes.

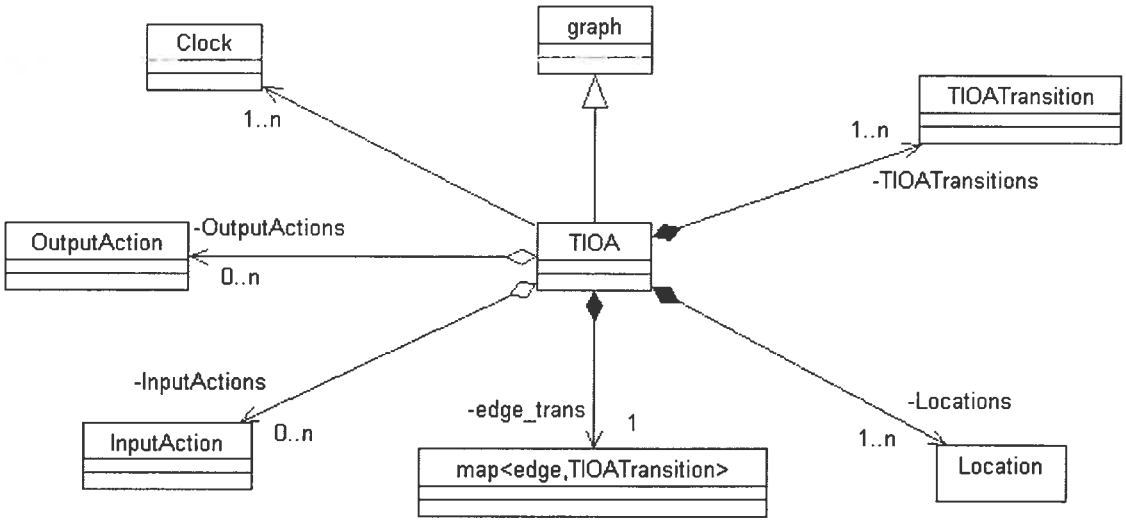

Figure 6.3 : Diagramme de classes d'un AEST

De la même manière, une instance de la classe TIOATransition est composée des instances des classes avec lesquelles elle est en relation d'agrégation dans le diagramme

de la figure 6.4. La classe 'InputAction' pour l'action d'entrée, la classe 'OutputAction' pour l'action de sortie et la classe 'ClockGuard' pour l'ensemble des contraintes d'horloges (ClockGuards). Chaque transition de l'AEST relie deux emplacements (The2Locations) et remet à 0 une ou plusieurs horloges (ClocksToReset).

Une contrainte d'horloge est composée d'une horloge du 'package' *TIOAView*, d'un opérateur (<, >, ≥, ≤) et d'une constante entière.

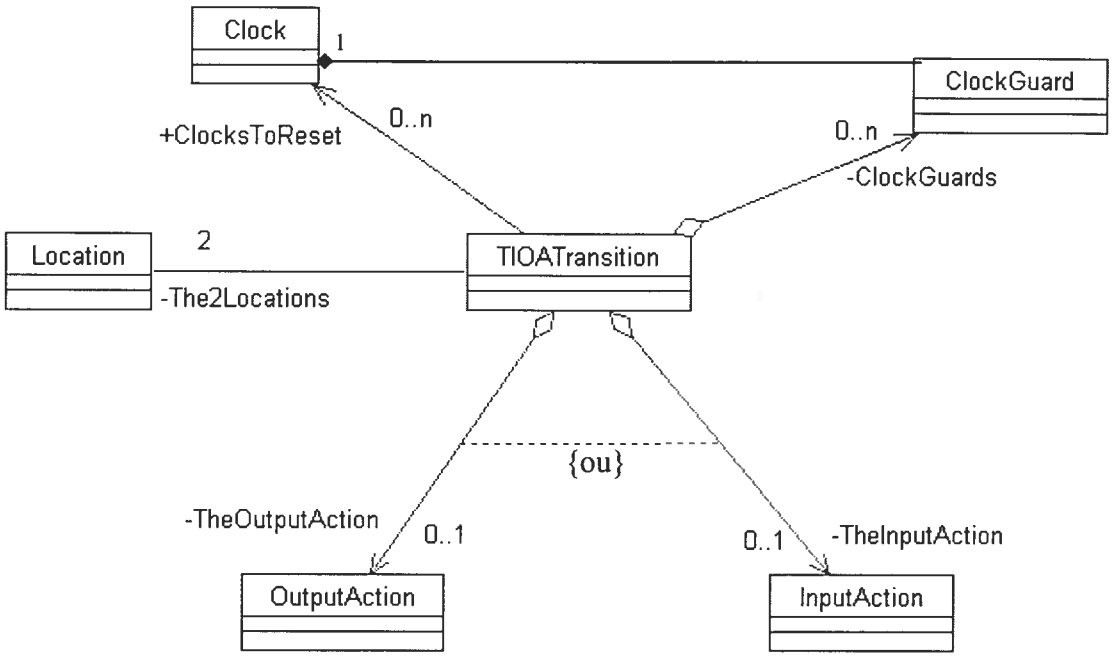

Figure 6.4 : Diagramme de classes d'une transition dans l'AEST

GridAutomata-NTFSMView : représente le diagramme de classes qui permettent de charger en mémoire la structure de données d'un automate de grille et de la transformer en MEFTNM. La classe 'Grid-Automata' représente un automate de grille. Elle est composée d'un ensemble d'états (States), d'un ensemble de transitions (GridAutTransitions) et elle hérite aussi de la classe 'graph'.

Chaque transition de l'automate de grilles (classe 'GridAutTransition') relie deux états et lui correspond un ou deux transitions dans l'AEST (classe 'TIOAtransition').

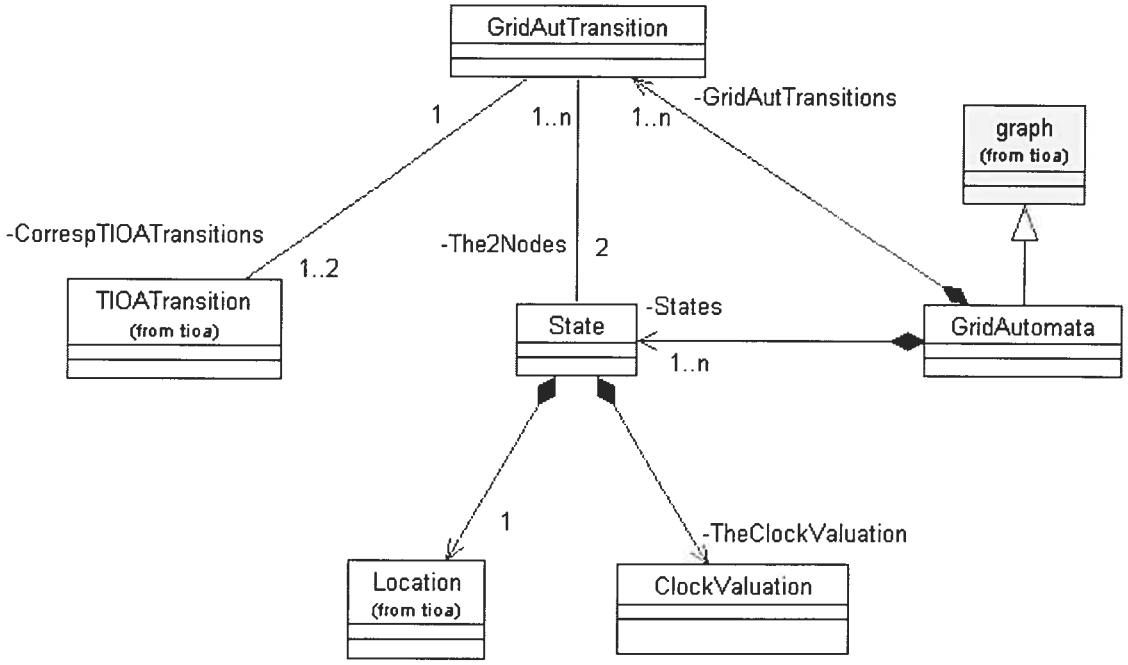

Figure 6.5 : Diagramme de classes d'un automate de grille/MEFTNM

Diagnostic-View : représente le diagramme de classes relatives aux structures de données générées lors du processus de diagnostic. La classe 'NTFSMSuspects' contient les méthodes qui permettent de générer les transitions suspectes au niveau la MEFTNM. Ces méthodes appliquent une suite de test (classe 'TestSuite') à la MEFTNM pour générer les ensembles de conflits (ConflictSets). L'intersection de ces derniers constitue les 'suspects'. Chaque ensemble de conflits est constitué d'une ou plusieurs transitions de la MEFTNM (ConflictSetTransitions) et il est calculé à partir de deux suites de sorties : la suite de sorties attendues (ObservedOutputSuite) et la suite de sorties observées (ExpectedOutputSuite).

La classe OutputSuite contient à son tour les primitives pour générer ces deux derniers ensembles en appliquant une suite de test (classe 'TestSuite') à la MEFTNM.

La classe 'TIOASuspects' utilise la classe 'ConflictSet' pour générer l'ensemble des transitions suspectes au niveau de l'AEST (Suspects). Elle génére aussi les machines mutantes (Mutants) correspondant à chaque transition suspecte.

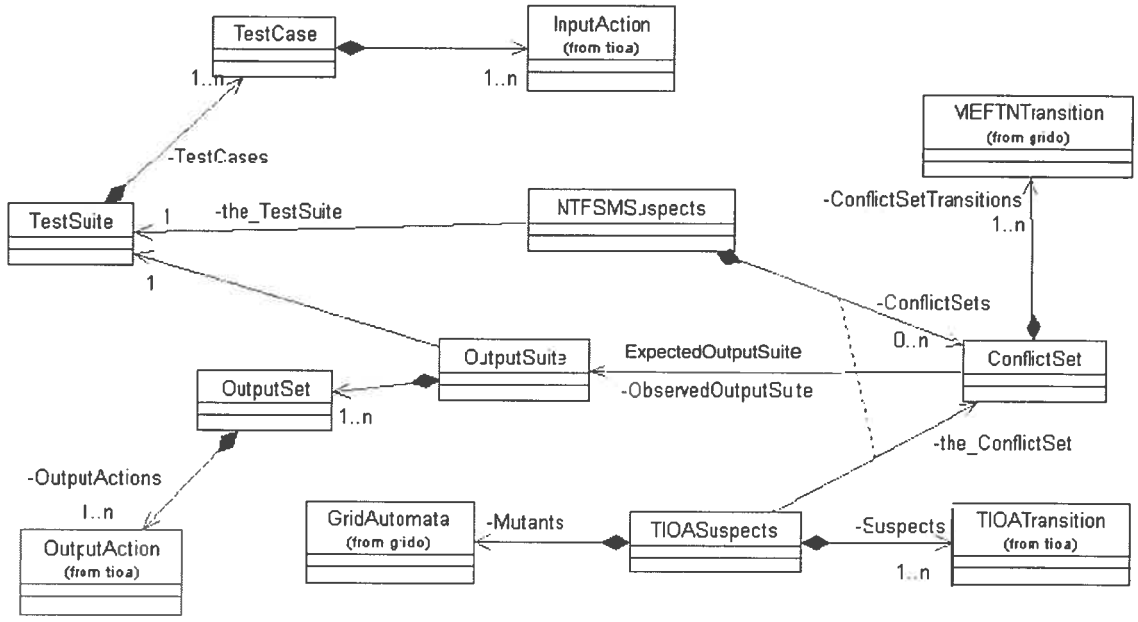

Figure 6.6 : Diagramme de classes pour la vue diagnostic

Pour des raisons de lisibilité, les attributs et les méthodes ne figurent pas dans les diagrammes de classes, l'Annexe 1 présente en détail les classes des différents diagrammes.

6.3.2 Résultats d'exécution

Nous présentons dans ce qui suit un exemple d'exécution de l'outil 'RTDiag' qui implémente la méthode proposée. Chaque figure présente une étape d'exécution de l'outil sur l'exemple illustratif du chapitre 5 (voir figures 5.1 et 5.2 qui représentent, respectivement, la spécification et l'implantation fautive).

Figure 6.7 : Chargement de la spécification et de l'implantation

Chargement des automates

L'acquisition des automates de l'implantation et de la spécification se fait à partir de deux fichiers textes qui doivent respecter une syntaxe bien définie (voir Annexe 2). Ces fichiers sont compilés et chargés sous une forme interne en utilisant la librairie GTL (figure 6.7).

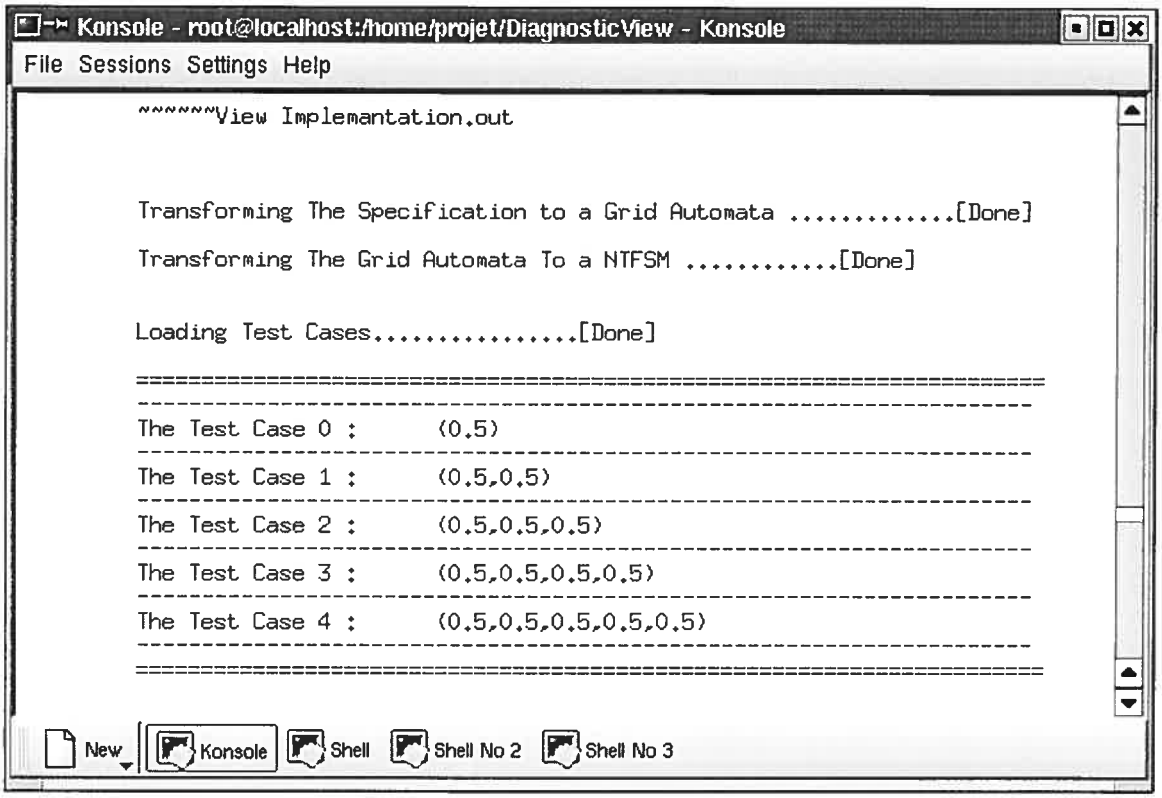

Figure 6.8 : Transformations de la spécification et chargement des cas de test

Transformation de la spécification et chargement de la suite de test

Dans cette étape, nous transformons la spécification en une structure interne d'automate de grille, puis en une MEFTN minimale. Les cas de test, générés à partir de la méthode Wp temporisée, sont ensuite chargés à partir d'un fichier externe (figure 6.8).

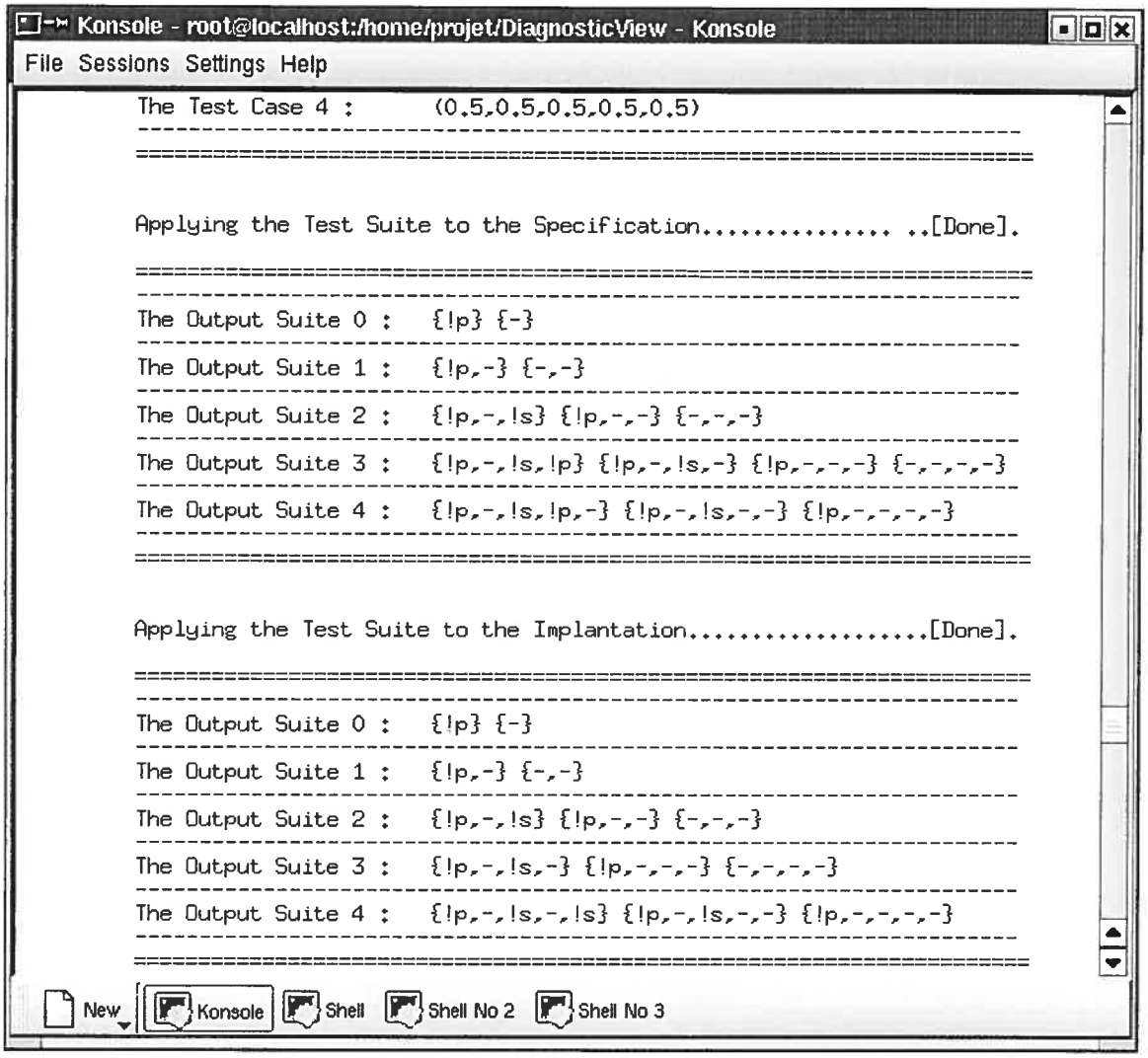

Figure 6.9 : Application des cas de test

Application des cas de test

Nous appliquons chaque cas de test à la spécification pour collecter la suite de sorties attendues et à l'implantation pour stimuler son comportement et observer la suite de sorties générées (figure 6.9).

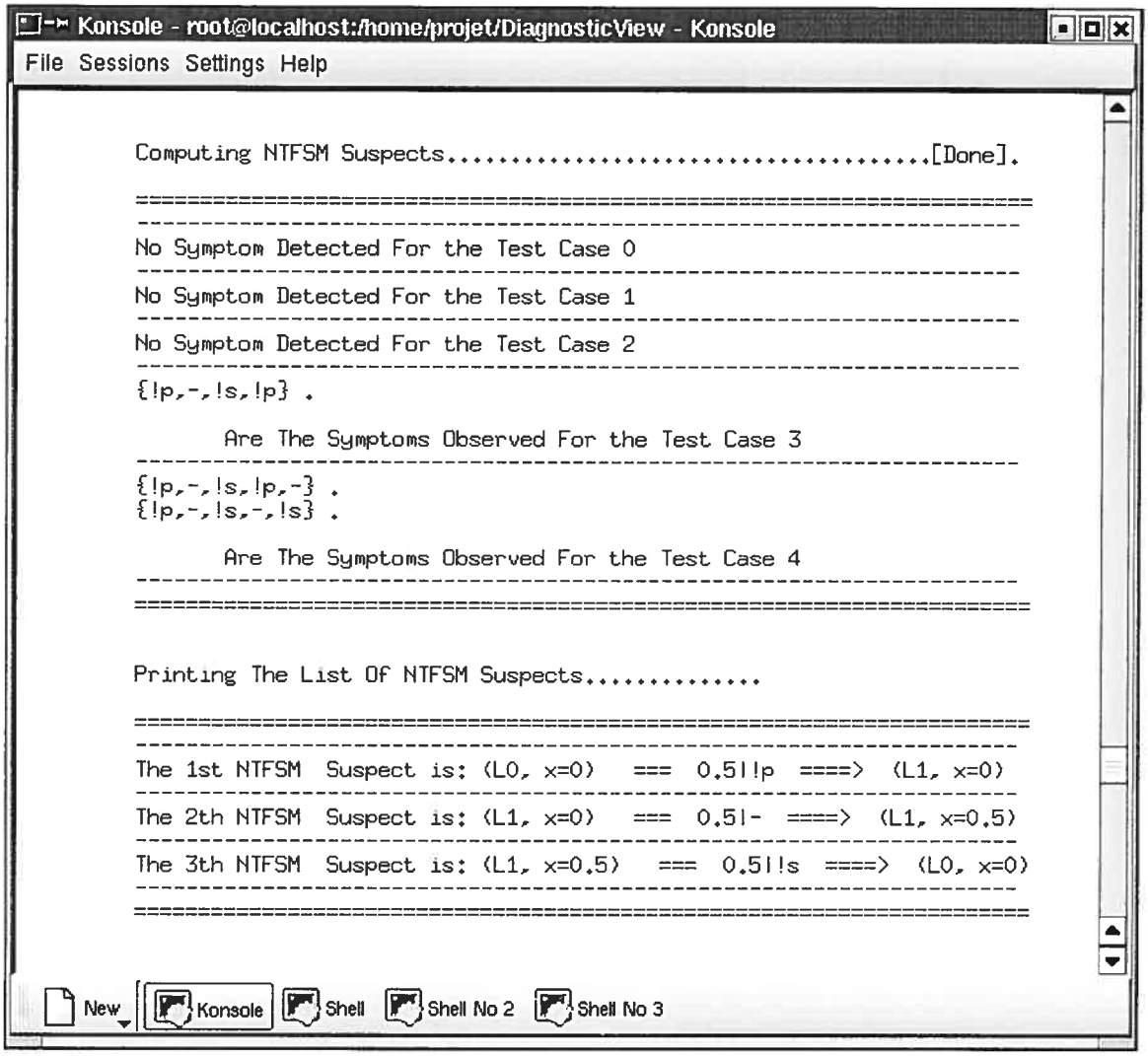

Figure 6.10 : Détection et analyse des symptômes

Détection des anomalies et génération des suspects

'RTDiag' compare la suite de sorties observées à celles attendues pour chaque cas de test et détecte les symptômes. Des anomalies sont observées pour les cas de test 3 et 4. L'algorithme d'analyse des symptômes a permis ici de localiser trois transitions suspectes dans la MEFTN (figure 6.10).

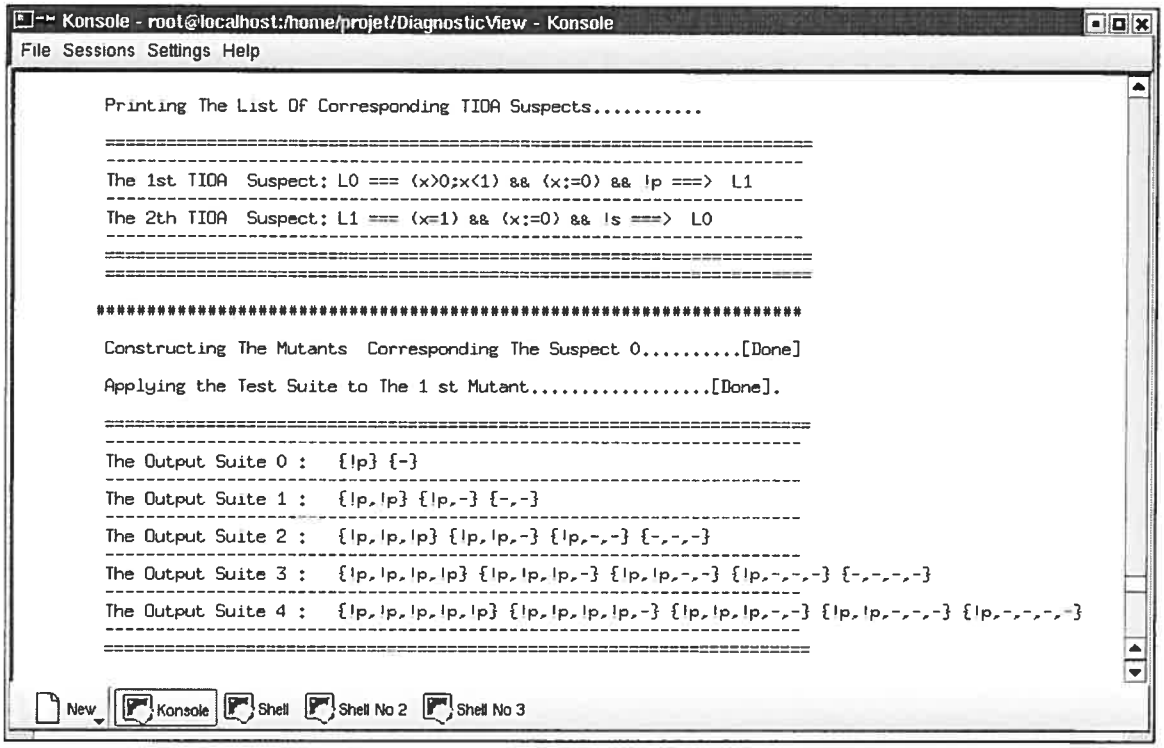

Figure 6.11 : Application des cas de test à la première machine mutante

Application des cas de test aux machines mutantes

Dans cette étape, nous appliquons les deux algorithmes proposés au chapitre 5 pour localiser les transitions suspectes au niveau de l'AEST (Figure 6.11). Ensuite, pour chacune de ces transitions, nous construisons les AESTs (machines mutantes) représentant éventuellement l'implantation et nous y appliquons les cas de test.

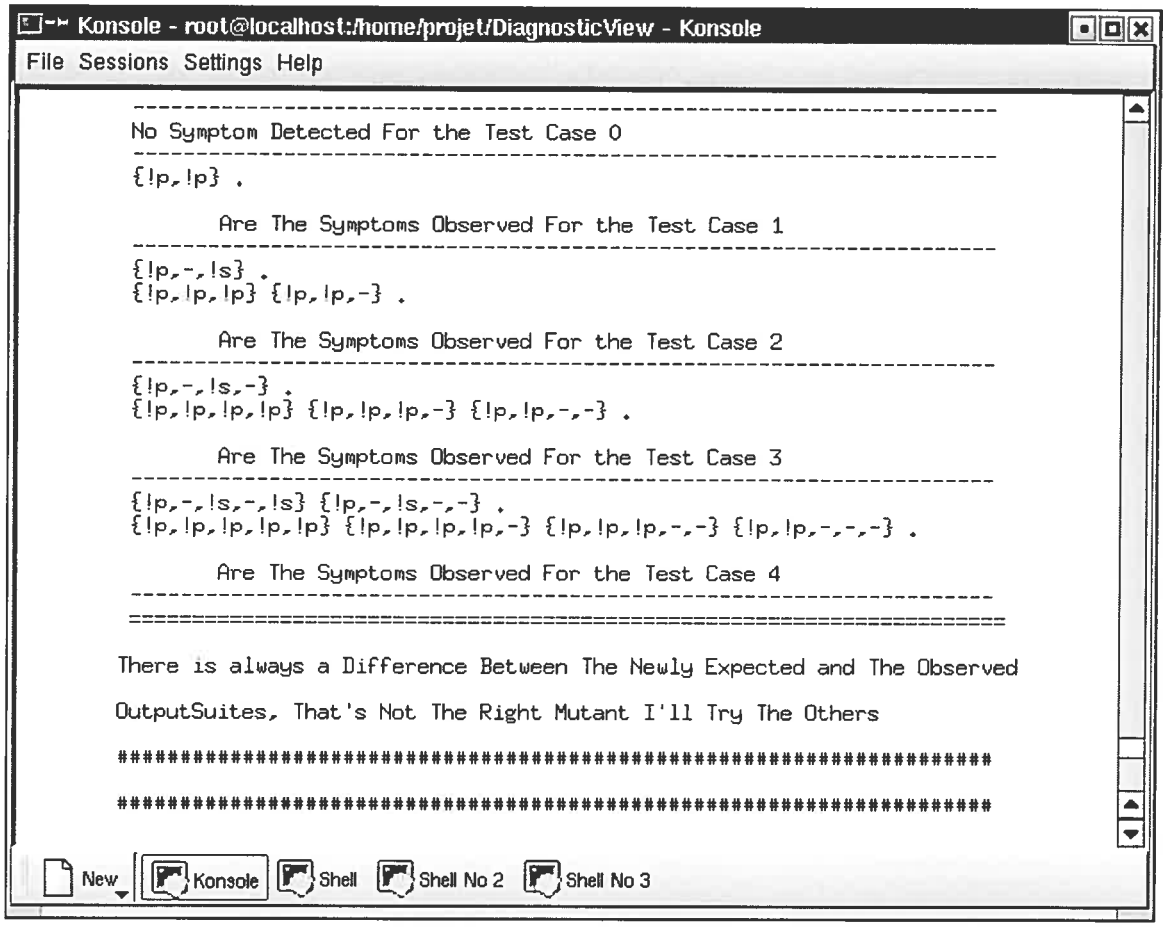

Figure 6.12 : Analyse des sorties de la première machine mutante.

Analyse des machines mutantes

Les sorties nouvellement collectées lors de l'application des cas de test sur la première machine mutante ne reflètent pas le comportement observé de l'implantation (figure 6.12). Les cas de test sont appliqués aux autres machines mutantes (figure 6.13) pour observer leurs comportements

D'après la figure 6.13, la machine mutante correspondant à la deuxième transition suspecte fournit les mêmes sorties observées que l'implantation pour tous les cas de test. Cette machine mutante est la seule à se comporter comme l'implantation. Nous pouvons alors conclure que la transition réellement fautive est la deuxième transition suspecte qui reste à l'emplacement l_1 au lieu de transférer à l'emplacement l_0 (figure 6.14).

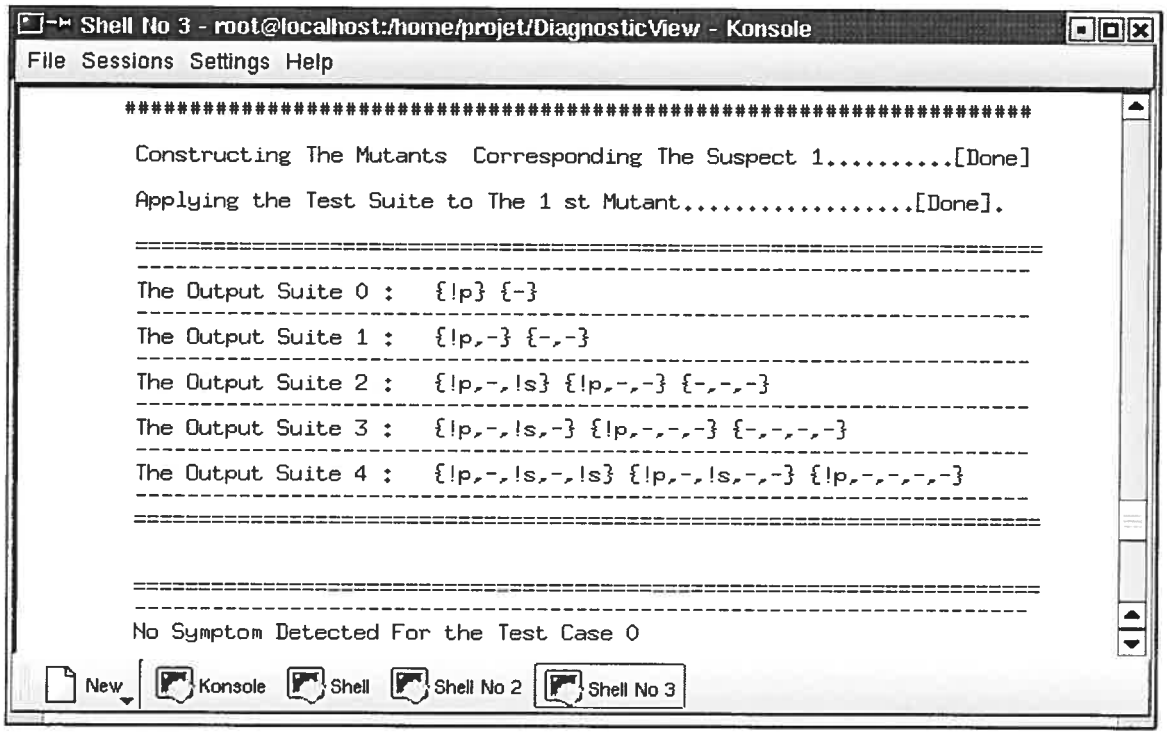

Figure 6.13 : Application des cas de test sur la machine mutante correspondant au deuxième transition suspecte.

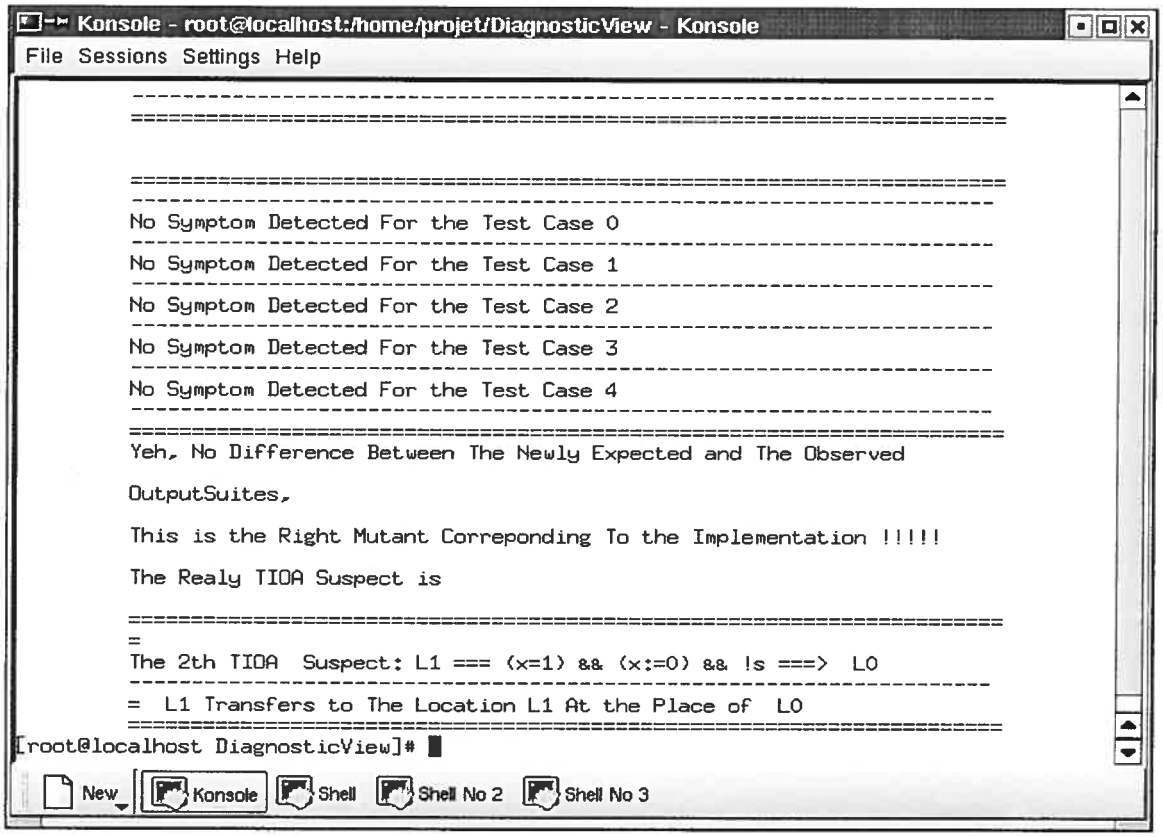

Figure 6.14 : Localisation de la transition réellement fautive

6.4 Conclusion

Dans ce chapitre, nous avons proposé une architecture conceptuelle orientée objet pour développer un outil de diagnostic temps réel que nous avons appelé 'RTDiag'. Le langage UML a été utilisé pour la modélisation des différentes composantes du système. Nous avons utilisé les 'STLs' ('Standard Template Library') conjointement avec la librairie 'GTL'. Cette dernière est devenue un standard pour les développeurs qui travaillent avec des structures complexes de graphes. Grâce au processus de développement choisi, notre application est facilement extensible.

Chapitre 7

Conclusion générale

Dans ce mémoire, nous avons développé une méthode pour le diagnostic des fautes de sortie et de transfert dans les systèmes temps réel modélisés par des automates à entrées sorties temporisées. Pour ce faire, nous avons commencé par introduire le sujet et présenter les différents concepts relatifs au processus du diagnostic. Nous avons présenté l'état de l'art du diagnostic basé sur les modèles formels de spécification. En particulier, nous avons abordé le diagnostic basé sur les MEFs et les MEFEs. Nous avons également présenté les modèles de spécification, les modèles de fautes, les méthodes de diagnostic et leurs couvertures ainsi que les méthodes de test qui y sont associées.

Vu le rôle primordial que la phase de test joue dans un processus de diagnostic, nous avons passé en revue les méthodes de génération de test à partir des modèles temporels. Pour chacune des méthodes présentées, nous avons expliqué les avantages, les inconvénients et les conditions d'application. Nous avons opté pour la méthode Wp temporisée pour générer des cas de test utilisés dans le processus de diagnostic, car cette méthode a l'avantage d'être générale et pratique avec une couverture complète de fautes. De plus, elle génère un nombre restreint de cas de test et elle est capable de détecter toutes les fautes temporelles dans une implantation modélisée par un AEST.

Notre contribution principale a consisté à développer une méthode de diagnostic des systèmes temps réel modélisés par des AESTs. Cette méthode permet de détecter et de localiser les fautes de sortie et de transfert dans ces systèmes. Elle consiste en plusieurs étapes principales.

Dans un premier temps, nous transformons l'AEST en automate de grille puis en MEFTN observable et minimale. L'objectif de cette étape est d'avoir une machine facilement testable, sur laquelle nous pouvons appliquer les séquences de test générées par la méthode Wp temporisée. Dans la deuxième étape, nous exécutons chaque cas de test sur la MEFTN pour générer les sorties attendues, puis sur l'implantation pour la stimuler et observer la suite de sorties. Dans la troisième étape, nous comparons les sorties observées à celles attendues. Si des symptômes sont détectés, nous générons un ensemble minimal de transitions suspectes de la MEFTN qui peuvent expliquer les anomalies dans le comportement de l'implantation. Dans la quatrième étape, nous analysons les symptômes relatifs aux différents cas de test pour dégager des hypothèses qui expliqueraient les erreurs détectées, et générer un ensemble réduit de transitions suspectes. Étant donné que la spécification de départ est sous forme d'AEST, nous avons proposé deux algorithmes de transformation pour identifier les transitions suspectes dans l'automate de grille, puis dans l'AEST de la spécification.

Ensuite, pour chacune de ces dernières transitions suspectes, nous construisons les AESTs, dits machines mutantes, représentant son comportement erroné. Nous appliquons les cas de test sur chacun de ces automates pour comparer les sorties nouvellement attendues à celles déjà observées de l'implantation. Seules les transitions suspectes, pour lesquelles les machines mutantes émettent les mêmes sorties observées, sont maintenues dans cette étape. Enfin, s'il y'a plusieurs machines mutantes qui représentent le comportement de l'implantation, nous générons des cas de test additionnels pour discriminer entre les candidats.

Un outil implantant la nouvelle méthode formalisée dans ce travail a été également développé. Un exemple d'exécution complet a été aussi présenté. La mise en œuvre de cet outil, appelé 'RTDiag', a été optimisée grâce à l'utilisation des librairies 'STL' et 'GTL'. Un emploi judicieux de ces deux librairies, la modélisation orientée objet et l'architecture conceptuel proposée nous ont permis de développer un outil de diagnostic facilement réutilisable pour des extensions futures.

Les travaux de ce mémoire peuvent faire l'objet de plusieurs extensions. Plus précisément, il serait intéressant de :

- Considérer que l'implantation peut avoir plusieurs fautes de sortie et/ou de transfert dans une ou plusieurs transitions.

- Utiliser des objectifs de diagnostic pour ne diagnostiquer que les parties critiques d'un système temps réel modélisé par un AEST, et réduire ainsi le nombre de transitions suspectes et de machines mutantes générées.

- Adapter et étendre notre approche pour le diagnostic des autres fautes définies dans le modèle de fautes temporel.

BIBLIOGRAPHIE

[ALUR 94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126: 183 – 235, 1994.

[BOCH 78] G.v Bochmann. Finite State Description of Communication Protocols. *Computer Networks*, Vol. 2, No. 4/5, 1978

[BOCH 91] G.v Bochmann, R. Dssouli, A. Das, M. Dubuc, A. Ghedamsi, G. Luo. Fault Models in Testing. *Proc. IFIP Intern. Workshop on Protocol Test Systems* (invite paper) pp. (II-17)-(II-32)

[BOUM 00] S. Boumaraf. Diagnostic des Protocoles de Communication Fondé sur les Automates à États Finis Étendus. *Mémoire de maîtrise*, Université de Montréal, 2000.

[BOUM 00] S. Boumaraf. Diagnostic des Protocoles de Communication Fondé sur les Automates à États Finis Étendus. *Mémoire de maîtrise*, Université de Montréal, 2000

[BOUR 99] C. Bourhfir. Génération Automatique des Cas de Test pour les Systèmes Modélisés par des Automates à États Finis Communicantes. *Thèse de doctorat*, Université de Montréal, 1999

[CERA 92a] K. Cerâns. Algorithmic Problems in Analysis of Real Time System Specifications. *Dr.sc.comp. thesis*, University of Latvia, Rïga, 1992.

[CERA 92b] K. Cerâns. Decidability of Bisimulation Equivalences for Parallel Timer Processes. In G. von Bochmann and D.K. Probst, editors, *Proccedings of the 4th International Workshop on Computer Aided Vérification,* Montréal, Canada, volume 663 of *Lecture Notes in Computer Science,* pages 302-315. Springer-Verlag, 1992.

[CLAR 97] 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.

[CLAR 96] D. Clarke. Testing RealTime Constraints. *PhD Thesis*, University of Pennsylvania, 1996.

[CHOW 78] T.s. Chow. Testing Software Design Modeled by Finite State Machines. *ITC*, Wachington, DC, USA, 1990, pp. 162-168

.[DAVI 82] R. Davis, H. Shrobe, W. Hamsher, K. Wieckert, S. Polit. Diagnosis Based on Description of Structure and Function, *Proceeding AAA1*, pp. 137-142, 1982

[DAVI 84] R. Davis. Diagnosis Reasoning Based On Structure and Behavior. *Artifical Intelligence 24 (3)*, pp. 347-410, 1984.

[DAVI 88] R. Davis, W. Hamscher. Model-Based Reasoning: Troublesshooting. *Artifical Intelligence, Shrobe H.E, The American Association for Artifical Intellignce(eds)*, pp. 297-346, Morgan Kaufman, 1988

[ELGH 02] K. El Ghazouani, M. Nourelfath, A. En-Nouaary et R. Dssouli. Diagnostic de Fautes de Transfert Simples dans les Systèmes Temps Réel Modélisés par des Automates à Entrées Sorties Temporisées. *Colloque Francophone de l'Ingénierie des Protocoles (CFIP)*, Montréal, Québec, Canada, Mai 2002.

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

[ENNO 01] A. En-Nouaary. Génération de Cas de Test pour les Systèmes Temps Réel Modélisés par des Automates à Entrées Sorties temporisées. *Thèse de*

Doctorat, Département d'informatique et de recherche opérationnelle, Université de Montréal, 2001.

[ENNO 02] A. En-Nouaary, R. Dssouli and F. Khendek. Timed Wp: Testing Real-time Systems. *IEEE Transactions on Software Engineering*, Vol. 28, No. 11, Novembre 2002.

[ENDE 97] A. En-Nouaary, R. Dssouli, and A. Elqortobi. Génération de Tests Temporisés. In *Proceedings of the* 6th *Colloque Francophone de l'ingénierie des Protocoles, HERMES, ISBN 2-86601-639-4,* 1997.

[FIJU 91] S. Fijuwara, 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

[GHED 93a] A. Ghedamsi. Diagnostic Tests for Protocol Implementations Modeled by Finite State Machines. *Thèse de doctorat*, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1993.

[GHED 93b] A. Ghedamsi, G.v Bochmann, R. Dssouli. Diagnosing Multiple Faults in Finite State Machines. *Technical Report 859*, Dept. IRO, Université de Montréal, January 1993, Accepted for IFIP 93.

[KLEE 87] J. de Kleer, B.C Williams. Diagnosing Multiples Faults. *Artifical Intelligence*, 32(1), 1987, pp. 97-130.

[LUOG 94] G. Luo, G. V. Bochmann and Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machine Using a Generalized Wp-Method. *IEEE Transactions on Software Engineering*, SE-20, N° 2: 149 – 162, 1994.

[LINF 93] F. Lin. Test Generation based on FSM Model with Timers and Counters. *Mémoire de maîtrise*, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1993

[MERL 79] P.M. Merlin. Specification and Validation of Protocols. *IEEE Trans, on Communications*, Vol. COM-27 No.11, 1979, pp. 1671-1679.

[MORE 90] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Trans, on Software Engineering*, Vol. 16, No 8, August 1990, pp. 844-857.

[MILN 80] Robin Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science,* vol. 92, 1980.

[MILN 89] Robin Milner. Communication and Concurrency. *Prentice Hall International*, 1989.

[MAND 95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Spécifications. *ACM Transactions on Computer Systems,* 13(4) pp. 365-398, November 1995.

[REIT 87] R. Reiter. A Theory of Diagnosis From First Principles. Artifical Intelligence 32(1), pp. 57-96, 1987.

[SCHW 82] R.I Schwartz, P.M. Melliar-Smith. From State Machine to Temporal Logic: Specifications Methods for Protocol Standards. *IEEE Trans, On Communications*, Vol. COM-30, No 12, 1982, pp. 2486-2496.

[SPRI 01] J. Springintveld, F. Vaadranger and P. Dargenio. Testing Timed Automata. *Theoretical Computer Science*, 254:225-257, 2001.

[VUON 90] S.T. Vuong and K.C. Ko. A Novel Approach to Protocol Sequence Generation. *IEEE Global telecom. Conference and exhibition*, San Diago, California, December 2 - 5, 1990, Vol. N° 3, 904.1.1 - 904.1.5.

Annexe 1

Implantation des classes du système

1. Package 'TIOAView'

class TIOA: public graph

```
{
  public:
    TIOA();
    void catchTIOA(char * file, const char * file1); //Construction du 'TIOA' à partir du
      // fichier de spécification
    vector<Clock> extractClocksToReset(string clocksToReset); // extraire les horloges
      // à remettre à zéro
    vector<ClockGuard> extractClocksGuard(string guardClocks); // extraire les
      // conditions sur les horloges
    void insertLocations(string loc1, string loc2, TIOATransition& trans); // ajouter les
      // emplacements à une fonction de transition
    void printTIOATransitions();
    TIOATransition catchTransition(edge ed); // retourne la fonction de transition
      // correspondante à l'arc du graphe ('graph' de GTL ) donné en argument
    float granularity(); // retourne la granularité de l'échantillonnage
    edge findEdge(string str1, string str2);
    bool findLocation(Location loc);
    void assign(edge theEdge, TIOATransition tr); // associe une fonction de transition
      // à  un arc du graphe ('graph' de la librairie  'GTL')
    ~TIOA();

 private:
   vector<Location>  Locations;
   vector<TIOATransition> TIOATransitions;
   vector<OutputAction> OutputActions;
   vector<Clock> Clocks;
   vector<InputAction> InputActions;
   map<edge,TIOATransition> edge_trans;

   vector<Location> getTheRestOfLocations(Location loc);
   const vector<Location>   get_Locations () const;
   void set_Locations (vector<Location>  value);
```

```
    const vector<TIOATransition> get_TIOATransitions () const;
    void set_TIOATransitions (vector<TIOATransition> value);
    const vector<OutputAction> get_OutputActions () const;
    void set_OutputActions (vector<OutputAction> value);
    const vector<Clock> get_Clocks () const;
    void set_Clocks (vector<Clock> value);
    const vector<InputAction> get_InputActions () const;
    void set_InputActions (vector<InputAction> value);
};

class InputAction
{
 public:
    InputAction(string iAction); // construction d'une action d'entrée
    void print();
    ~InputAction();

 private:
    string name;

    const string get_name () const;
    void set_name (string value);

};

class OutputAction
{
 public:
    OutputAction(string oAction); // construction d'un symbole de sortie
    void print();
    ~OutputAction();

 private:
    string name;

    const string get_name () const;
    void set_name (string value);
};

class Location
{
 public:
    Location(string location); // construction d'un emplacement
    void print();
    ~Location();
```

```
 private:
    string name;

    const string get_name () const;
    void set_name (string value);
};

class Clock
{
 public:
    Clock();
    void make (string name,int theMax); // une horloge avec un nom et une valeur
    // maximale
    void print(string mes);
    void reset();
    ~Clock();

 private:
    string name;
    int maxValue;
    float currentValue;

    void setNextValue(float gran);
    const string get_name () const;
    void set_name (string value);
    const int get_maxValue () const;
    void set_maxValue (int value);
    const float  get_currentValue () const;
    void set_currentValue (float value);
};

class ClockGuard
{
 public:
    ClockGuard(Clock cl, string op, int cst); // une condition sur l'horloge cl, op est
        // l'opérateur (<,<=,>,>=) de comparaison et l'entier cst est la borne inférieure
        ou la borne supérieure
    void print();
    bool verifiedBy( Clock clock); // retourne si vrai ou faux l'horloge 'Clock' vérifie la
        // condition de l'objet courant
    ~ClockGuard();
```

```
 private:
    string oper;  // l'opérateur
    int constant; // la constante
    Clock TheClock;  // l'horloge

    const Clock get_TheClock () const;
    void set_TheClock (Clock value);
    const string get_operator () const;
    void set_operator (string value);
    const int get_constant () const;
    void set_constant (int value);
};
```

class TIOATransition

```
{
 public:
    TIOATransition(string); // construction une transition de l'AEST à partir d'un 'string'
    void print();
    void print_in_file(char* file); // écriture dans le fichier 'file' des éléments de la
        // transition courante
    ~TIOATransition();

 private:
    vector<Location> The2Locations; // les deux emplacements de la transition
    OutputAction TheOutputAction; // l'action de sortie
    vector<Clock> ClocksToReset; // les horloges à remettre à zéro
    vector<Clock> AllClocks; // tous les horloges de l'AEST
    InputAction TheInputAction; // l'action d'entrée
    vector<ClockGuard> ClockGuards; // les conditions sur les horloges

    const vector<Location> get_The2Locations () const;
    void set_The2Locations (vector<Location> value);
    const OutputAction get_TheOutputAction () const;
    void set_TheOutputAction (OutputAction value);
    const vector<Clock> get_ClocksToReset () const;
    void set_ClocksToReset (vector<Clock> value);
    const vector<Clock> get_AllClocks () const;
    void set_AllClocks (vector<Clock> value);
    const InputAction get_TheInputAction () const;
    void set_TheInputAction (InputAction value);

};
```

2. Package 'GridAutomata-NTFSMView'

class GridAutomata : public graph

```
{
  public:
    GridAutomata(TIOA ta);    // construction d'un automate de grille à partir d'un AEST
    State  makeInitialState(TIOA ta); // retourne l'état initial
    bool findState(vector<State> accesStates, vector<State> traitedStates, State &state);
    bool valVerifyClockGuards( ClockValuation cval, vector<ClockGuard> clkGuards);
        // retourne si oui ou non la valuation d'horloges 'cval' vérifie les conditions
        // clkGuards
    void assignGrTransition(edge theEdge, GridAutTransition tr); // associe une
        // fonction de transition à un arc de l'objet 'graph'.
    void assignState(node theNode, State theState); // associe un état à un noeud du
        // graphe
    State catchState(node nd); // retourne l'état correspondant au noeud donné en
        // argument
    GridAutTransition catchTransition(edge ed);
    void  GA_MEFTN();
     ~GridAutomata(TIOA ta);

  private:
    vector<GridAutTransition> GridAutTransitions;
    vector<State> States; // l'ensemble des états de l'automate
    TIOA tioa; // l'AEST correspondant
    map<node,State> Node_State;  // la structure permettant le 'maping' Noeud/État
    map<edge,GridAutTransition> Edge_GridTrans; // la structure permettant le maping
        // Arc/Fonction de transition
    const vector<GridAutTransition> get_GridAutTransitions () const;
    void set_GridAutTransitions (vector<GridAutTransition> value);
    const vector<State> get_States () const;
    void set_States (vector<State> value);
    void set_tioa (TIOA value);
};
```

class State

```
{
  public:
    State(ClockValuation theClockVal, Location theLocation); // un état est construit à
        // partir d'emplacement et d'une valuation d'horloges
    void print();
    void print(ostream& outs);
    bool operator==(const State &right) const;
     ~State();
```

```
 private:
    Location TheLocation;
    ClockValuation TheClockValuation;

    const Location get_TheLocation () const;
    void set_TheLocation (Location value);
    const ClockValuation get_TheClockValuation () const;
    void set_TheClockValuation (ClockValuation value);
};

class GridAutTransition
{
 public:
    GridAutTransition(State state1, State state2, InputAction ip, OutputAction op);
        // construction d'une transition de l'état 'state1' à l'état 'state2' sur une action
        // d'entrée ou de sortie
    GridAutTransition(State state1, State state2, float granul); // construction d'une
        // transition explicitant un écoulement de temps de 'granul' unités
    void print();
    void print(int flag);
    void print(ostream& outs, int flag);
    bool isInputTrans(); // retourne vrai si la transition est sur une action d'entrée
    bool isOutputTrans(OutputAction & opt); // retourne vrai si la sortie opt est celle
        // fournie par la transition en cours
    bool isGranulTrans();
    void makeEmptyOp();
    bool hasTheInput(InputAction Ip);
    bool operator==(const GridAutTransition &right) const;
     ~GridAutTransition();

private:
    vector<State> The2Nodes; // les deux états, l'état de départ et celui d'arrivée
    InputAction theInputAction; // l'action d'entrée
    OutputAction theOutputAction; // l'action de sortie
    float Granul; // non vide si la transition explicite un écoulement de temps
    vector<TIOATransition> CorrespTIOATransitions;

    const vector<State> get_The2Nodes () const;
    void set_The2Nodes (vector<State> value);
    const float get_Granul () const;
    void set_Granul (float value);
    const InputAction get_theInputAction () const;
    void set_theInputAction (InputAction value);
    const OutputAction get_theOutputAction () const;
    void set_theOutputAction (OutputAction value);
```

```
    const vector<TIOATransition>  get_CorrespTIOATransition () const;
    void set_CorrespTIOATransition (vector<TIOATransition>  value);
};
```

3. Package 'DiagnosticView'

class NTFSMSuspects

```
{


 public:
   NTFSMSuspects();
   vector<GridAutTransition> computeNTFSMSuspects(vector<OutputSuite>
                                opSuites1, vector<OutputSuite> opSuite 2, int sze)
      // calcule les transitions suspectes dans la MEFTN à partir de la suite de sorties
      // attendues et la suite de sorties observées
   vector<GridAutTransition> intersection(vector<GridAutTransition> vec1,
                                vector<GridAutTransition> vec2)
      // retourne l'intersection de deux ensembles de transitions

 private:
    vector<GridAutTransition> MNTFSMSuspects;

    const vector<GridAutTransition> get_MNTFSMSuspects () const;
    void set_MNTFSMSuspects (vector<GridAutTransition>  value);
};
```

class ConflictSet

```
{
  public:
    ConflictSet(OutputSuite *waitedFor, OutputSuite *Observed); // calcul l'ensemble de
        // conflits correspondant une suite de sorties attendues et celle de sorties
          observées
    void addConflictTransition(GridAutTransition gaTrans);
      //  ajouter une transitions suspecte
    bool findOutputSet(vector<OutputSet> waitedForSuite, OutputSet setToFind);
      // retourne vrai si l'ensemble de sorties 'setToFind' se trouve dans la suite de
     // soties  attendues
    bool isIncluded(OutputSuite firstOpSuite, OutputSuite secondOpSuite);


    vector<OutputSet> computeDiff(OutputSuite firstOpSuite, OutputSuite
                          secondOpSuite);
       // calcul la différence entre deux suites de  sorties
```

```
    vector<GridAutTransition> computeMaxCommonPrefixes(OutputSuite checkSet,
                                        OutputSet oneDiff, int flag);
        // calcul le plus grand préfixe commun entre l'ensemble 'oneDiff' et la suite de
        // sorties checkSet et retourne les transitions correspondant à ce préfixe
    ~ConflictSet();

  private:
    OutputSuite *ObservedOutputSuite; // la suite de sorties observées
    OutputSuite *WaitedForOutputSuite;  // la suite de sorties attendues
    vector<GridAutTransition>   ConflictSetTransitions; // ensemble de conflits

    const OutputSuite * get_ObservedOutputSuite () const;
    void set_ObservedOutputSuite (OutputSuite * value);
    const OutputSuite * get_WaitedForOutputSuite () const;
    void set_WaitedForOutputSuite (OutputSuite * value);
    const vector<GridAutTransition>  get_ConflictSetTransitions () const;
    void set_ConflictSetTransitions (vector<GridAutTransition>  value);

};

class OutputSuite
{
  public:
    OutputSuite(GridAutomata *ga,TestCase tc,int i); // une suite de sorties est obtenue
        // aprés l'application du cas de test 'tc' à l'automate 'ga' transformée en MEFTN
    void  applyThis(node nd, vector<InputAction> ips, int i, OutputSet opSet);
    void  addOutputSet(OutputSet opSet); // ajoute l'ensemble de sorties opSet
    ~OutputSuite();

private:
    GridAutomata *the_GridAutomata;
    TestCase  the_TestCase;
    vector<OutputSet>   OutputSets;

    const GridAutomata*  get_the_GridAutomata () const;
    void set_the_GridAutomata (GridAutomata * value);
    const TestCase  get_the_TestCase () const;
    void set_the_TestCase (TestCase  value);
    const vector<OutputSet>  get_OutputSets () const;
    void set_OutputSets (vector<OutputSet>  value);

};
```

```
class OutputSet
{
 public:
    OutputSet(); // construction d'un ensemble de sorties
    void addOutput(OutputAction op) ;
    void addWaitedForTransition(GridAutTransition gt);
    void print();
     bool operator==(const OutputSet &right) const;

 private:
    vector<OutputAction> OutputActions;
    vector<GridAutTransition> WaitedForTransitions

    const vector<OutputAction> get_OutputActions () const;
    void set_OutputActions (vector<OutputAction> value);
    const vector<GridAutTransition> get_WaitedTransitions () const;

};

class TestCase
{

 public:
    TestCase(string tc); // construction d'un cas de test
    void print(int l);
    ~TestCase();

 private:
    vector<InputAction>  InputActions;

    const vector<InputAction>  get_InputActions () const;
    void set_InputActions (vector<InputAction>  value);

};

class TestSuite
{
 public:
     TestSuite(); // construction d'une suite de test
     void catchSuite(char* file);
      ~TestSuite();

 private:
    vector<TestCase> TestCases;
```

```
    const vector<TestCase> get_TestCases () const;
    void set_TestCases (vector<TestCase> value);
};
```

Autres fonctions utilisées dans le main :

```
vector<TIOATransition> computeTIOAsuspects(vector<GridAutTransition>
                NTFSMSusps)
    // calcul des transitions suspectes de l'AEST correspondant aux transitions
    // suspectes au niveau de la MEFTNM
bool findGridAutTransition(vector<GridAutTransition> vec1, GridAutTransition gat)
    // retourne vrai la transition 'gat' est dans le vecteur de transitions 'vec1'
bool findTIOATransition(vector<TIOATransition> vec1, TIOATransition tioatrans)
vector<TIOA>  computeMutant(TIOA tioa, TIOATransition tioaTrans,
                vector<Location>& faultLocations)
    // calcul les AESTs mutantes relatives aux emplacements fautifs 'faultLocations'
    //  auquels la transition 'tioaTrans' paurrait transférer
int findRightMutant(vector<TIOA> mutants, vector<OutputSuite> opSuites, TestSuite
                TS)
    // permet de trouver l'AEST qui représente l'implantation en réappliquant la suite de
    // test 'TS' à tous les machines mutantes 'mutants'  et en comparant les sorties
    // nouvellement attendues avec les sorties 'opSuites' déjà observées de l'implantation.
```

Annexe 2

Structure d'un fichier de spécification et de son implantation erronée

Nous présentons la structure d'un exemple de fichier de spécification :

```
Clocks
  x  2
  y  2
End;


  L0        ?On          (x:=0)              NULL              L1
  L1        !Out          NULL               (x>=1;x<=2)       L1
  L1        ?On          (x:=0;y:=0)         (x<1;y<2)         L0
```

Le premier bloc permet de déclarer l'ensemble des horloges, utilisées par l'AEST de la spécification, ainsi que leurs valeurs maximales.

Le reste du fichier explicite les différentes transitions entre les nœuds de l'automate :

La première colonne indique que le nœud de départ, la deuxième explicite l'événement d'entrée ou de sortie. Ensuite, la troisième colonne spécifie les affectations à réaliser si la transition est exécutée. La quatrième colonne explicite les conditions de la transition et enfin la cinquième colonne indique le nœud d'arrivée.

Un exemple d'implantation erronée de cette spécification peut avoir la structure suivante :

```
Clocks
  x  2
  y  2
End;


  L0        ?On          (x:=0)              NULL              L0
  L1        !Out          NULL               (x>=1;x<=2)       L1
  L1        ?On          (x:=0;y:=0)         (x<1;y<2)         L0
```

Nous remarquons que la première transition transfert vers L0 ou lieu de L1.