2m11.2781.6

Université de Montréal

Diagnostic des protocoles de communication fondé sur les automates à états finis étendus

Par

Samira Boumaraf

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

En vue de l'obtention du grade de

Maître es science (M.Sc.)

En informatique

Février, 2000

© Samira Boumaraf

QA
76
U54
2000
v.015

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Diagnostic des protocoles de communication fondé sur les automates à états finis étendus

Présenté par:

Samira Boumaraf

A été évalué par un jury composé des personnes suivantes:

Président-rapporteur: El Mostapha Aboulhamid
Directeur de recherche: Rachida Dssouli
Membre du jury: Houari Sahraoui

Mémoire accepté le 18 février 2000

Sommaire

Les vérifications préalables effectuées sur un protocole de communication durant son cycle de développement ne garantissent pas toujours une implémentation du protocole sans erreurs, de nombreuses fautes sont introduites pendant la phase d'implémentation.

Pour résoudre ce problème d'incompatibilité, il devient nécessaire de trouver des techniques qui permettront d'analyser ces protocoles, pour détecter les éventuelles erreurs existantes, assurer la conformité des implémentations par rapport à leurs spécifications afin de rétablir leur bon fonctionnement.

Le diagnostic vise l'identification et la localisation des fautes détectées par les tests de conformité. Dans le domaine des protocoles de communication, beaucoup d'efforts sont consacrés au test de conformité. L'objectif de ce test est de vérifier que les exigences de la spécification du protocoles sont satisfaites par son implémentation.

Nous considérons le diagnostic comme une étape de l'ingénierie des logiciels de communication, elle est nécessaire au débogage et à la maintenance des systèmes. Il est dans ce cas intimement lié à l'activité du test car l'une de ses phases est fondée sur l'analyse de traces d'exécution des cas de test.

Dans ce mémoire, nous avons passé en revue certaines techniques de diagnostic fondées sur un modèle et en particulier celles appliquées dans le domaine de protocoles de communication. Nous avons proposé une approche de diagnostic fondée sur l'usage des automates à états finis étendus par les variables. Notre contribution concerne le développement d'une méthode de diagnostic pour le flux de données de l'automate. Nous avons également développé un outil EMFDT ("Extended Multiple fautes Diagnostic Tool") qui implante la méthode de diagnostic proposée. EMFDT est un outil automatique de diagnostic. Il permet, à partir d'une spécification de protocole (écrite dans un langage spécifique) et d'une trace d'exécution de suite des tests, de générer des diagnostics à propos des transitions fautives d'une implémentation sous test.

Table des matières

Sommaire ... *i*

Table des matières ... *ii*

Liste des figures ... *iv*

Liste des tables ... *v*

Remerciements ... *vi*

1 Introduction ... *1*

1.1 Définition du domaine ... 1

1.2 Problème et Motivation ... 3

1.3 Objectifs et contribution de notre travail ... 4

2 Les approches de diagnostic ... *7*

2.1 Les concepts généraux en diagnostic ... 7

2.1.1 Génération des candidats diagnostic ... 7

2.1.2 Discrimination entre les candidats ... 7

2.1.3 Le modèle de fautes ... 8

2.2 La méthode de Davis ... 8

2.3 La théorie de Reiter ... 10

2.4 L'approche de deKleer et Williams ... 12

3 Le test des protocoles de communication ... *16*

3.1 Test de Conformité ... 16

3.2 Test de conformité dans le cadre de l'OSI ... 16

3.3 Conception des suites de tests ... 19

3.3.1 Introduction et définitions ... 19

3.3.2 Sélection de tests ... 19

3.3.3 Sélection des tests fondée sur les automates à états finis ... 21

4 Le diagnostic des protocoles de communication ... *24*

4.1 Les spécifications de protocoles ... 24

4.2 Les approches de diagnostic pour les protocoles de communication ... 27

4.2.1 Les approches "système expert" ... 28

4.2.2 Les approches classiques ... 28

4.2.3 Les approches de diagnostic fondé sur un modèle ... 29

4.3 Les travaux en diagnostic ... 32

4.3.1 Diagnostic de protocoles ... 32

4.3.2 Tests de protocoles ... 35

5 choix du formalisme de représentation et définitions ... *40*

5.1 Choix du formalisme de représentation ... 40

5.2 Le modèle de représentation ... 41

5.3 Définitions[Weyu 85] 42

5.4 Modèle de fautes 42

6 Un algorithme général de diagnostic 45

7 L'algorithme et techniques proposés 47

8 Conception du système de diagnostic: "Extended Multiple Fautes Diagnostic Tool (EMFDT) " 52

8.1 Architecture conceptuelle du système 52

8.2 Exemple 53

9 Implantation du système EMFDT 63

9.1 Représentation interne du système 63

9.2 Représentation graphique des structures de données 63

9.3 Le modèle de données du système 65

9.4 Le langage de développement 67

9.5 Implantation du modèle de données de l'application 70

9.6 Implantation des modules de l'application 70

9.7 Interface du système 70

Conclusion 82

Références 84

ANNEXE A 90

ANNEXE B 91

ANNEXE C 95

Liste des figures

Figure 1 : Interface entre un système de résolution de problèmes et un ATMS ________ 13

Figure 2 : Exemple d'un automate à états finis ________ 25

Figure 3 : Exemple d'automate à états finis étendu ________ 26

Figure 4 : Exemple de graphe d'AEFE. ________ 43

Figure 5 : Graphe de dépendances des transitions de l'exemple de la figure 4 ________ 49

Figure 6 : Décomposition en modules de l'outil "EMFDT" ________ 53

Figure 7 : Implémentation erronée de la spécification de la figure 4 ________ 54

Figure 8 : Trace d'exécution (Comportement observé de l'IST) ________ 56

Figure 9 : Sorties extraites ________ 56

Figure 10 : Symptôme du cas de l'exemple 5 ________ 58

Figure 11 : Modèle de données du système ________ 66

Liste des tables

Table 1: procédure de génération des dépendances des transition ______________ *51*

Table 2 : Liste des hypothèses ______________ *59*

Remerciements

Je tiens à remercier ma directrice de recherche, le professeur Rachida Dssouli pour avoir eu confiance en moi au tout début de mon inscription en maîtrise en me proposant ce sujet. Je la remercie également pour sa compréhension durant la période de ma grossesse, et de son support moral et financier tout au long de ma maîtrise.

J'aimerais également remercier le groupe téléinformatique pour l'ambiance sympathique.

Aussi, je remercie mon cher mari qui m'a toujours soutenu et encouragé et sans qui ce travail n'aurait pas abouti. Je remercie mon adorable fille Wissam Lydia qui est venue au monde alors que je rédigeais ce mémoire, je la remercie aussi pour toute la joie et le bonheur qu'elle m'apporte jour après jour.

Je remercie toute ma famille qui a toujours eu confiance en moi, qui m'a encouragé, et qui m'a donné le goût de faire des études et d'aller toujours plus loin.

Je remercie également tous mes amis, en particulier Rachid Azzi pour ses discussions fructueuses et le temps qu'il m'a consacré.

Un grand merci à tout le personnel aimable et dévoué de la bibliothèque Math-info.

Enfin, je tiens à exprimer ma reconnaissance envers la Chaire industrielle HP de l'université de Montréal pour sa contribution financière.

1 Introduction

Ce projet a été initié par le centre de test de Hewlett-Packard dans le cadre de la chaire industrielle détenue par le professeur G.V. Bochmann. Ce travail vient compléter le travail qui a été réalisé par S. Htite dans le cadre du diagnostic fondé sur les automates à états finis [Htit 97].

Ce chapitre introductif donne un aperçu du problème qui sera abordé dans ce mémoire. Il sera question dans ce chapitre d'introduire la notion du diagnostic, ses concepts de base et le rôle d'un tel processus pour les protocoles de communication.

1.1 Définition du domaine

Qu'est ce qu'un protocole de communication?

Un protocole est un ensemble de règles établies que deux ou plusieurs entités, distantes doivent implanter afin de pouvoir communiquer ou échanger des informations. l'organisation de standardisation internationale (OSI, en englais "ISO") est à l'avant-garde du développement et de la promotion des normes pour les protocoles de communication. Dans son modèle de référence pour l'interconnexion des systèmes ouverts (ISO, "OSI") [Knig 88], un système de communication est décomposé en sept couches hiérarchiques. Chaque couche à l'exception de la première, est basée sur le service offert par la précédante. Une couche est caractérisée par un ensemble de fonctions. Pour réaliser ces fonctions, un protocole est défini pour chaque couche.

Le processus de développement d'un protocole passe par plusieurs étapes, il débute par l'étape de spécification et se termine par l'étape d'implantation et de tests. Une spécification de protocole est la description qui défini les interactions entre les entités pour la réalisation des fonctions attendues par le protocole afin de fournir les services d'une couche donnée. Pour spécifier des protocoles, une variétés de méthodes basées sur différents modèles ont été proposées dans la littérature. On trouve des méthodes orientées vers les transitions d'états, telles que celles qui se basent sur les automates à états finis [Boch 78] et les réseaux de Petri [Merl 79] et des méthodes qui se basent sur les langages formels et la logique temporelle [Schw 82, Miln 80].

Une implémentation de protocole doit être testée par rapport à sa spécification afin de vérifier la conformité du protocole par rapport à ce qui a été spécifié. Cette phase est appelée le test de conformité, plusieurs recherches ont été faites et plusieurs méthodes ont été développées pour de tels tests [Boch 83, Rayn 87, Sari 89].
Une spécification de protocole présente plusieurs aspects: Un aspect qui régit le contrôle et un aspect de données, ces deux aspects peuvent être traités séparément dans le contexte du test de conformité.
La partie données, qui consiste à tester les paramètres d'entrées, les primitives de sortie, et les variables locales, a été traitée par un très petit nombre de chercheurs, utilisant certaines analyses de flot de données[Ural 87, Vuon 89] et des approches de tests fonctionnels [Sari 87, Sari 89]. Pour la partie contrôle, plusieurs méthodes ont été proposées, elles produisent des séquences de tests pour détecter les fautes d'une implémentation sous test (IST).

Le diagnostic

L'activité du diagnostic consiste en la recherche de la ou les raisons qui expliquent l'apparition de fautes ou d'incohérences dans un système. Il se rapport à l'action de détection, de localisation et de correction des fautes observées.
Le diagnostic est un sujet bien étudié dans différentes applications telles que
les circuits logiques, les systèmes mécaniques complexes et la médecine. Il suit principalement trois étapes :

1. l'application de données de test au système sous observation,
2. la conclusion que les résultats de test diffèrent de ceux attendus,
3. la présentation des explications concernant la cause du problème.

Dépendant du système à diagnostiquer et sa représentation, différentes méthodes et techniques sont utilisées. On distingue deux classes de diagnostic [Khen 90]. La première classe appelée « Diagnostic expérimental », principalement utilisée en médecine et dans d'autres domaines similaires, elle ne sera pas discutée dans ce rapport. La deuxième classe, celle qui nous intéresse, appelée « Diagnostic du premier principe » ou « Diagnostic basé sur un modèle » [Stru 90] ou aussi «Diagnostic basé sur la structure et le comportement » [Davi 88, Klee 87]. Dans ce type de diagnostic, il est nécessaire de

savoir comment est supposé fonctionner le système sous diagnostic, afin de déterminer pourquoi il ne fonctionne pas correctement.

Dans le domaine des protocoles de communication, l'approche utilisée est celle fondée sur un modèle, plus généralement, les Automates à États Finis (AEF).

1.2 Problème et Motivation

Les vérifications préalables effectuées sur un protocole de communication durant son cycle de développement ne garantissent pas toujours une implémentation du protocole sans erreurs, en effet de nombreuses fautes sont introduites pendant la phase implémentation. Ces erreurs sont principalement dues :

- au fait que plusieurs implémentations du même protocole peuvent résulter suite à une mauvaise interprétation de la spécification du protocole,
- à l'intervention humaine,
- chaque protocole offre souvent une série d'options qui peut produire des incompatibilités,
- l'accroissement continu de la complexité des protocoles,
- spécifications de protocoles incomplètes.

Pour résoudre ce problème d'incompatibilité, il devient nécessaire de trouver des techniques qui permettront d'analyser ces protocoles, pour détecter les éventuelles erreurs existantes, assurer la conformité des implémentations par rapport à leurs spécifications afin de rétablir leur bon fonctionnement.

Durant la dernière décennie, la génération systématique des séquences de tests pour les protocoles de communication a été un domaine de recherche très actif. De nouvelles méthodes de sélection de test ont été développées pour le test de la conformité des implémentations de protocoles par rapport à leurs spécifications. Ces méthodes sont surtout basées sur le modèle des machines à états finis. Elles produisent des séquences de tests destinés à détecter des fautes dans les transitions d'une implémentation sous test (IST). Néanmoins, l'application de ces séquences ne permet pas nécessairement la localisation des fautes détectées. La détection d'une ou plusieurs fautes dans une implémentation n'étant pas suffisante pour le rétablissement du bon comportement

spécifié, il devient important de trouver des moyens qui permettent la localisation de ces fautes.

Un deuxième problème complémentaire au test est celui du diagnostic. Il vise l'identification et la localisation des fautes détectées par les tests de conformité. Dans le domaine des protocoles de communication, beaucoup d'efforts sont consacrés au test de conformité. L'objectif de ce test est de vérifier que les exigences de la spécification du protocoles sont satisfaites par son implémentation. Dans ce cas la spécification du protocole joue le rôle d'une référence pour la sélection de tests et pour le processus de diagnostic. Une telle spécification de référence est disponible dans notre cas.

Deux approches sont possibles pour la localisation des fautes, la première consiste à analyser la trace des tests, alors que la deuxième permet de localiser la faute à partir de la construction de la machine qui fournit le comportement observé.

Le but de ce travail est donc de résoudre le problème du diagnostic dans le domaine des protocoles de communication. Nous contribuons à apporter une bonne prise en charge des fautes de protocoles en adoptant comme formalisme de base un automate à états finis étendu afin de faciliter le diagnostic et représenter la richesse des informations présentes dans une spécification de protocoles.

Nous considérons le diagnostic comme une étape de l'ingénierie de logiciels de communication, elle est nécessaire au débogage et à la maintenance des systèmes. Il est dans ce cas intimement lié à l'activité du test car l'une de ses phases est fondée sur l'analyse de traces d'exécution des cas de test.

1.3 Objectifs et contribution de notre travail

Le travail que nous apportons par le biais de ce mémoire concerne une approche de diagnostic basée sur un modèle d'automates à états finis étendu, et la réalisation d'un outil pratique pour le diagnostic des protocoles de communication. Cet outil aura l'avantage d'être générique, flexible, fiable et offrira la capacité d'expliquer les fautes détectées dans les implémentations de protocoles dans le cadre du test de conformité.

Nous sommes dans le cas d'une entité unique qui représente l'IST, pour laquelle nous fournissons les entrées et nous observons les sorties, nous supposons que les observations sont fiables et qu'elles commencent à partir de l'état initial.
Nous apportons un complément en traitant les actions et les conditions de garde relatives aux transitions. Nous utiliserons pour cela les dépendances des transitions, introduites par Chanson [Chan 93]. Nous assimilons les séquences des transitions aux déclarations d'un programme pour lesquelles on doit vérifier les valeurs des variables et leur évolution.
Selon le formalisme utilisé pour décrire la spécification, une variétés de techniques peuvent être choisies pour dériver des suites de tests ou pour effectuer le processus de diagnostic. Dans de nombreux cas, une distinction est faite entre l'aspect contrôle et l'aspect données du système à tester. Le flot de contrôle est souvent décrit par un AEF, qui sert de modèle de référence pour le test et le diagnostic. Cependant pour des systèmes réels, le test du contrôle doit être complété par le test de l'aspect données, souvent traduit par un AEFE.
Nous nous intéressons au diagnostic de l'aspect données du système, représenté par un AEFE. Nous supposons que l'aspect contrôle du système a déjà été testé par l'outil Multiple Fautes Diagnostic Tool (MFDT) [Htit 97] basé sur un modèle d'AEF et qui permet de localiser toutes les fautes de transfert et de sorties contenues dans l'implémentation du protocole. Nous sommes concernés par la vérification des paramètres opérant sur les variables d'états, les prédicats appliquées aux transitions, ce qui induit une notion de contexte d'états de transitions.
Dans ce travail, nous proposons donc une étape complémentaire au test, le diagnostic. Ce diagnostic sera effectué dans le cas où l'exécution de la suite de tests détecte la présence d'un comportement erroné du système. Nous supposons que pour chaque faute de l'IST, il existe un cas de test qui détecte cette faute.

Méthodologie pour l'analyse des résultats de test et le diagnostic

Nous nous intéressons principalement au diagnostic basé sur un modèle. L'idée principale de ce type de diagnostic est de savoir comment doit se comporter le système pour déterminer pourquoi il ne se comporte pas correctement.

Pour se rapprocher de la réalité où les systèmes peuvent contenir plusieurs fautes, nous développons une méthode qui permet le diagnostic des fautes multiples dans des systèmes représentés par des automates à états finis étendus. Un outil est développé pour le diagnostic des fautes contenues dans la partie données du protocole.

Pour ce diagnostic nous supposons la disponibilité du système réel (l'implémentation) qui peut être observé et son modèle (la spécification) à partir du quel nous pouvons prédire le comportement du système.

Le système et son modèle sont supposés avoir les mêmes composants et la même structure.

Les observations d'entrées/sorties montrent comment le système se comporte, tandis que les sorties attendues indiquent comment le système est supposé se comporter. Les différences entre ce qui est observé et ce qui est attendu, appelées symptômes, indiquent des différences entre le modèle et son système. Ce processus de comparaison des observations et des résultats attendus est appelé "analyse des résultats de tests".

Pour expliquer les symptômes observés, un processus de diagnostic est nécessaire. il consiste principalement en deux étapes : la génération des candidats susceptibles d'expliquer les symptômes observés, et la discrimination entre les candidats [Klee 87].

2 Les approches de diagnostic

Dans le but de développer une méthodes de diagnostic pour les protocoles de communication, nous avons jugé nécessaire d'étudier les méthodes de diagnostic existantes dans d'autres domaines. Nous avons étudié les méthodes en intelligence artificielle, dans le domaine des circuits logiques et du génie logiciels. Parmi ces méthodes les plus importantes sont présentées dans ce chapitre.

2.1 Les concepts généraux en diagnostic

Le diagnostic se défini comme la différence minimale entre le système et son modèle. les différences devraient expliquer toutes les anomalies observées sur le système. Pour obtenir les différences entre le système et son modèle, le processus de diagnostic exécute deux taches principales: la génération des candidats et la discrimination entre les candidats.

2.1.1 Génération des candidats diagnostic

A partir des symptômes observés et du modèle, le processus de diagnostic déduit des hypothèses sur les éventuelles causes des fautes observées. Ces hypothèses expliquent la différence entre le système et son modèle. Chaque hypothèse indique une fautes dans un ou plusieurs composants 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 candidats plus d'une fois, et il est optimal s'il génère uniquement des candidats minimaux et non des sur-ensembles de candidats.

2.1.2 Discrimination entre les candidats

A la fin du processus de générations des candidats, il en résulte 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 tests supplémentaires, appelés tests de distinction ("Distinguishing test") [Davi 82, Davi 84, Gene 84]. La deuxième technique consiste quant à elle à choisir de nouveaux points d'observation dans le système sous diagnostic afin de le rendre testable.

2.1.3 Le modèle de fautes

Il arrive souvent dans des système durant le processus de diagnostic, qu'un composant erroné ait des comportements autres que ceux prévus. Pour cette raison, plusieurs travaux ont introduit la notion de modèle de fautes. Cette notion a été introduite la première fois dans les systèmes de diagnostic GDE+ [Stru 89] et Sherlock [Klee 89]. Le modèle de fautes exprime toutes les fautes possibles de chaque composant du système. Il aide à réduire considérablement l'espace des candidats.

2.2 La méthode de Davis

Davis a étudié le problème du diagnostic, il traite le cas d'une faute simple dans les circuits [Davi 82, Davi 84, Davi 88]. La méthode se base sur le comportement et la structure du modèle. Le comportement de chaque composant du circuit est exprimé par des contraintes qui décrivent la relation entre les entrées et les sorties du système. Il distingue deux types de contraintes. Le premier type, appelé contraintes de simulations, décrit comment les composants sont supposés fonctionner. Le deuxième type, appelé contraintes d'inférence, est utilisé pour déduire les valeurs des variables afin de vérifier que ces valeurs ne sont pas en contradiction avec l'ensemble des contraintes.

Procédure de diagnostic

Le processus de diagnostic se fait en deux étapes. La première étape est la génération des candidats diagnostic, qui consiste à déterminer tous les composants suspectés fautifs et qui expliquent les observations du système. La deuxième étape quant à elle sert à réduire l'espace des candidats générés lors de la première étape, en appliquant au système certaines séquences de tests supplémentaires.

Génération des candidats diagnostic

Cette procédure se fait en trois étapes :

- **Étape 1 : Identification des symptômes**

La suite de tests sélectionnée est appliquée en même temps aux circuits pour observer le comportement du système sous diagnostic, et aux réseaux de contraintes pour collecter le comportement attendu du système. Une comparaison entre ce qui est observé et ce qui est attendu permet de détecter les conflits et permet alors de générer ce qu'on appelle des symptômes.

- **Étape 2 : Générations des suspects**

Pour chaque symptôme, on remonte le chemin dans le réseau de contraintes afin d'identifier tous les composants participant à produire le symptôme concerné. Chaque composant appartenant à ce chemin est considéré comme «suspect». A la fin, une fois que tous les symptômes ont été considérés, il en résulte plusieurs ensembles de suspects, ils sont alors combinés pour produire tous les suspects qui pourront expliquer tous les symptômes.

- **Étape 3 : Sélection des candidats**

Pour chaque suspect S, l'ensemble des contraintes reliées à S sont retirées du réseau de contraintes décrivant le système. Les sorties observées sont insérées dans le réseau réduit, les contraintes du nouveau réseau sont alors exécutées. Si le réseau reste dans un état consistant, ce qui implique une absence de contradictions entre les contraintes du nouveau réseau, le suspect S est confirmé comme un candidat diagnostic. Sinon, si des contradictions sont obtenues après l'exécution des contraintes, le suspect S ne peut être confirmé et sera éliminé de la liste des candidats. Un exemple détaillé peut être trouvé dans [Ghed 93a].

- **La discrimination entre les candidats**

Sous l'hypothèse de faute simple, seulement un candidat de ceux générés représente le candidat fautif. Pour isoler ce candidat, de nouveaux tests doivent être sélectionner et

appliqués au système. Davis [Davi 88] et Shirley [Shir 83] ont proposé une approche pour discriminer entre les candidats.

2.3 La théorie de Reiter

Reiter a développé une théorie générale de diagnostic basée sur un modèle [Reit 87]. Sa théorie permet de déterminer tous les diagnostics capables d'expliquer les conflits entre le comportement prédit et celui observé d'un système donné. Il propose un algorithme qui s'applique aussi bien dans le cas d'une faute simple que dans le cas de fautes multiples. Le procédé du diagnostic se base sur la structure et le comportement du modèle.
Le système sous diagnostic est représenté par une paire (SD, Composants) où :

- SD est la description du système exprimée en logique du premier ordre,
- Composants, les composants du système, un ensemble fini de constantes.

Une **observation** du système est un ensemble fini d'expressions logiques du premier ordre. Le système sur lequel des observations OBS ont été réalisées s'exprime par :
(SD, Composants, OBS)

Le **diagnostic** du système SD est défini comme l'ensemble minimal D de composants tels que :

$$SD \cup OBS \cup \{\neg AB(c) \mid c \in Composants - \Delta\} \cup \{AB(c) / c \in D\}$$

Est consistant. AB est un prédicat qui indique que le composant c se comporte anormalement.
En d'autres termes, l'expression du diagnostic indique qu'une explication logique peut-être trouvée pour les observations OBS du système, quand tous les composants fonctionnent normalement, mis à part ceux qui appartiennent à D.

La procédure du diagnostic

Avant de définir la procédure de diagnostic nous devons définir ce qu'est un «**hitting set** » .

Un « hitting set » pour un ensemble C de collection d'ensembles est un ensemble H tel que :

$$H \subseteq \bigcup_{S \in C} S \quad tel \quad que \quad H \cap S \neq \emptyset$$

H est minimal si aucun sous ensemble de H ne constitue un « hitting set » pour C.

La procédure de diagnostic se base sur la détermination des « hitting sets » minimaux pour les ensembles de conflits, définis comme suit :

Si le système contient des fautes, les observations seront en conflit avec ce que la description du système prédit. La prédiction du système est que tous les composants fonctionnent correctement, ce qui s'exprime dans cette théorie par :

SD U OBS U{¬AB(c1),..... ¬AB(ck)}.

Or si les observations sont en conflits, ceci implique que :

SD U OBS U{¬AB(c1),..... ¬AB(ck)} est inconsistant.

L'ensemble {c1,....., ck} est appelé ensemble de conflits.

A partir des ensembles de conflits et des « hittig sets », Reiter détermine le diagnostic du système.

Théorème : D, un ensemble de composants, est un diagnostic pour (SD, Composants, OBS) si et seulement si D est un « hitting set » minimal pour la collection des ensembles de conflits du système (SD, Composants, OBS).

Les ensembles « hitting sets » minimausx sont donnes par un algorithme qui construit un arbre appelé « hitting set tree » et qui a les propriétés suivantes :

Pour un ensemble C de collection d'ensembles non vides :

1- la racine de l'arbre est libellée par ✔ si C est vide, sinon par n'importe quel ensemble de la collection dans C.

$$H \cap S = \emptyset$$

2- pour chaque nœud n de l'arbre T, H(n) est constitué par l'ensemble des labels des chemins à partir de la racine jusqu'au nœud n. Le label du nœud n peut être n'importe quel ensemble S de C tel que :

Si cet ensemble existe le label de n est S et pour tous élément e de S, n a un successeur, ne, relié par un arc libellé par e. Si un tel ensemble S n'existe pas alors le label de n sera ✔.

Reiter a proposé un algorithme pour construire le plus petit arbre possible, générant les « hitting sets » minimaux.

Reiter déclare que pour chaque noeud n libellé par ✔ dans l'arbre T, H(n) est un « hitting set » pour l'ensemble des conflits C.

- **L'algorithme de diagnostic**

L'algorithme procède en deux étapes :

1- Il génère l'arbre des « hitting sets » T pour la collection des ensembles de conflits du système (SD, Composants, OBS),

2- Il retourne l'ensemble {H(n)/ n est un nœud de T libellé par ✔ } qui constitue l'ensemble de tous les diagnostic pour(SD, Composants, OBS).

2.4 L'approche de deKleer et Williams

de Kleer et Williams ont proposé une procédure de diagnostic basé sur un modèle. Cette procédure a été implantée dans un système appelé « General Diagnostic Engine (GDE) ». Dans leur approche, le diagnostic des fautes multiples est considéré et le comportement du système est décrit par des contraintes.

GDE se base principalement sur un autre système appelé ATMS (Assumption-based Truth Maintenance System). Avant de décrire GDE, nous devons décrire l'ATMS.

ATMS : Un ATMS est un système de maintien de vérité, il est utilisé comme un sous-système qui maintient la compatibilité des inférences générées par un autre système de

raisonnement. Il est chargé de stocker les informations relatives à la cohérence des différentes hypothèses et leurs implications. Il est vu comme un outil pour les systèmes de raisonnement comme illustré par la figure 1.

Le contexte d'un ATMS est un contexte logique ou certaines assomptions sont valides, un exemple d'assomptions peut être qu'un composant spécifique du système est supposé fonctionner correctement.

L'information est stockée dans l'ATMS sous forme d'entités appelées nœuds. Le système fournit à l'ATMS des conclusions et leurs inférences directes. Un nœud est alors créé pour chaque inférence, le nœud «Faux» est un nœud spécial qui correspond à une contradiction. Plusieurs concepts s'associent au nœud, ces concepts sont :

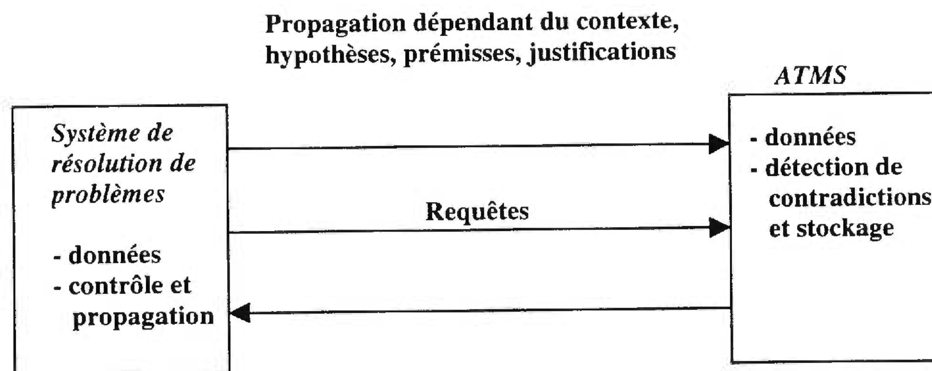


Figure 1 : Interface entre un système de résolution de problèmes et un ATMS[1]

- Une justification est une inférence fournie par le système, elle relie le nœud dérivé par l'inférence aux notes de maintient telles que les pré-conditions de l'inférence.
- Une hypothèse représente la décision du système à croire en la validité d'une inférence.
- Un environnement est représenté par l'ensemble des hypothèses à partir desquelles le nœud est induit utilisant les justifications existantes. Un nœud «Vrai» est absolu et n'a pas besoin d'hypothèses, il a un environnement vide. On appelle par «No-good»

[1] Exemple tiré de la thèse de [Ries 93a]

un environnement inconsistant, un environnement pour un nœud «Faux». Un contexte est formé par l'ensemble des hypothèses d'un environnement qui est consistant avec tous les nœuds dérivés de ces hypothèses utilisant les justifications existantes.

- Un label pour un nœud est l'ensemble des environnements minimaux (un environnement est minimal s'il ne contient pas des environnements comme des sous-ensembles) où les « No-goods » sont éliminés. Il représente la disjonction de l'ensemble des pré-conditions du nœud. Si le label est vide, alors les justifications et les hypothèses existantes ne permettent pas de dériver le nœud, il est considéré comme un fait. L'inférence contradictoire qui s'applique aux nœuds « Faux » est caractérisée par un label qui ne peut se combiner d'une manière consistante.

Exemple : Soit un multiplicateur M et soit X = 6, la valeur de sortie de M. La configuration associée à cet exemple serait un nœud X= 6, un label {M} et une justification {M, A=3, B=2, Multiplication} où A et B sont les entrées du système et ou M est supposé fonctionner correctement.

L'algorithme GDE

L'algorithme GDE fonctionne comme suit. Les observations sont propagées à travers les contraintes décrivant le système, les résultats de la propagation sont stockés dans l'ATMS. Quand les données sont en conflits, les assomptions sur lesquelles ces données sont basées sont considérées inconsistantes et globalement en conflits. Ces assomptions sont ajoutées à l'environnement du nœud «Faux».

Un exemple très connu basé sur un circuit composé de multiplicateurs et d'additionneurs est donné dans [Klee 87] et résumé succinctement dans [Ries 93a].

F est un nœud qui correspond à la sortie du multiplicateur, il a été créé par propagation des entrées A, B, C et D à travers le circuit qui représente le modèle correct. Il est prédit que F a une valeur 12 quand les composants A1, M1 et M2 sont supposés fonctionner correctement et quand les entrées observées sont A=3, B=2, C=2 et D=3. Ceci donne :

Nœud 1 : (F=12, {{A1, M1, M2}}, { A=3, B=2, C=2 et D=3...,})

A1 représente l'hypothèse que le composant additionneur A1 fonctionne correctement, et M1 et M2 sont des assomptions similaires pour les deux multiplicateurs.
F est observé avec une valeur 10, un autre nœud est alors créé :

Nœud 2 : (F=10, {{}}, {observations})

Les nœuds Observés de l'ATMS sont des nœuds observations, toutes les observations sont supposées fiables et sont alors considérées comme des prémices. Les nœuds contenant les données de la propagation dans le modèle sont appelés des propagations.
Comme on peut voir les valeurs de F dans le nœud et le nœud 2 sont contradictoires, F ne peut pas avoir la valeur 10 et 12 en même temps, si on suppose que l'observation est fiable, et que les composants du système fonctionnent correctement alors on en conclut que l'environnement {A1, M1, M2}du nœud 1 est un « No-good » et devient un membre de l'environnement du nœud « Faux » et le label du nœud 1 devient {}.
L'ensemble des conflits est utilisé pour déterminer l'ensemble des candidats diagnostic. Dans GDE, un candidat diagnostic est un ensemble d'assomptions D qui doivent être fautives. Chaque membre de D correspond à un composant qui peut être fautif. Par exemple un candidat diagnostic {A1}, signifie que nous devons abandonner l'hypothèse originelle concernant le fonctionnement correcte de A1 et qu'aucune dépendance liée à A1 n'est correcte. A ce moment les propagations n'induiront pas de contradiction.
Chaque candidat est créé par la sélection d 'au moins un composant de chaque conflit, afin d'expliquer chaque comportement fautif. Par exemple si nous savons que les ensembles {A1, M1, M2} et {A1, A2, M1, M3}sont tous les deux des environnements « No-goods », les candidats possibles seraient :

CANDIDATS = {{A1}, {M1}, {M2, A2}, {M2, M3}}

Cet ensemble est minimal dans le sens où l'ajout d'un candidat en plus serait redondant .
Pour discriminer entre les candidats générés à la première étape du diagnostic, de Kleer et Williams considèrent des observations en un nouveau point dans le circuit, ils se basent sur des probabilités pour définir le prochain point d'observation. Ces nouvelles observations permettront de réduire l'espace des candidats et d'isoler le vrai candidat d'une manière optimale.
L'algorithme termine lorsqu'un candidat est jugé certains (forte probabilité).

3 Le test des protocoles de communication

3.1 *Test de Conformité*

Durant le cycle de développement d'un protocole de communication, des erreurs peuvent être présentes et peuvent persister tout au long du cycle de développement. Plus on avance vers la phase d'implémentation plus le coût de correction d'une erreur dans la spécification augmente d'une façon considérable. Pour cela des méthodes de vérification des spécification des protocoles ont été développées afin de localiser les erreurs le plus tôt possible dans le cycle de développement et prévenir leur propagation. Cependant ces vérifications effectuées ne garantissent pas l'obtention d'une implémentation complètement correcte. Il est donc important de tester cette implémentation pour s'assurer qu'elle fonctionne conformément à sa spécification de référence.
Des approches ont été développées pour produire des tests. Parmi les plus connus nous citons:

- **Les tests à boite blanche**

Ce sont des tests de vision interne et ils sont bâtis à l'aide de la structure interne du code de l'IST. Ils sont habituellement effectués par des personnes ayant l'accès au code de l'IST

- **Les tests à boite grise**

Ce sont des tests intermédiaires, ils sont bâtis à l'aide de la structure interne de l'IST, sans tenir compte du code de celle-ci.

- **Les tests à boite noire**

Ce sont des tests réalisés à partir de la spécification du protocole, ils ne tiennent pas compte de la structure interne et du code de l'IST. Ils ont une vision interne et globale. Ils sont souvent réalisés dans des centres de tests spécialisés.

3.2 *Test de conformité dans le cadre de l'OSI*

Dans le cadre des systèmes ouverts, toutes les implémentations du protocoles du niveau N des différents systèmes, doivent être conformes à la spécification de la norme qui

définit le protocole. En particulier, une norme contient normalement une section, appelée clause de conformité, qui définit toutes les propriétés que chaque implémentation doit satisfaire pour être considérée conforme.
Quatre types de tests de conformité ont été identifiés selon ce qu'ils peuvent offrir comme indication sur la conformité de l'IST par rapport à sa spécification de référence:

- **Tests d'interconnexion de base**

Détermine si l'IST est capable d'établir une connexion avec le système de tests et que les éléments principaux du protocole sont réalisés, et ce avant d'entamer d'autres tests plus poussés. Ces tests sont appropriés:

- pour la détection d'erreurs sévères de non conformité,
- comme premier filtre avant d'entamer des tests plus coûteux.

Cependant, les tests d'interconnexion de base sont inappropriés:

- comme base pour établir qu'une implémentation est conforme,
- comme moyen d'arbitrage pour déterminer les causes d'échec de communication.

Les tests d'interconnexion de base forment généralement un petit sous-ensemble d'une suite de tests complète.

- **Tests de capacité**

l'énoncé de conformité ("protocol implémentation conformance statement") [ISO 9646] décrit les capacités de l'implémentation: la classe du protocole, les options, etc. ainsi que des contraintes statiques de conformité concernant ces capacités.
Le but des tests de capacité est de vérifier que l'IST supporte effectivement les capacités citées dans l'énoncé de conformité de l'IST.
Les tests de capacités peuvent être utilisés pour:

- vérifier la validité de l'énoncé de conformité de l'IST,
- vérifier que les règles statiques de conformité sont satisfaites,

- permettre une sélection efficace d'autres tests de conformité pour une IST particulière.

Les tests de capacité forment aussi un sous-ensemble d'une suite de tests complète.

- **Tests de comportement**

Ces tests concernent les règles dynamiques de conformité. On vérifie que l'IST satisfait ces règles comme elles sont définies dans la norme du protocole. On essaie, dans la limite du possible, de tester une IST de la foçon la plus complète. le nombre de combinaison des événements et de leurs occurrences dans le temps sont infinis, les tests de comportement ne peuvent être exhaustifs.

Les tests de comportement sont appropriés:

- pour être considérés comme base pour établir la conformité d'une IST, tant que les autres types de tests ou l'usage de l'implémentation ne prouvent pas le contraire.

Ils sont inappropriés:

- pour la résolution de problèmes survenus durant l'usage de l'implémentation ou dans le cas où d'autres tests ont signalé un aspect de non conformité même si la suite de tests a été satisfaite.

- **Tests pour la résolution de question de conformité**

Le but de ces tests est d'obtenir des réponses précises et définitives pour certaines question de conformité. Ceci est atteint par l'application des tests à un aspect restreint de l'IST et pour des situations préalablement identifiées. Par exemple, vérifier, lors du développement de l'implémentation, que certaines propriétés ont été correctement implantées, ou examiner, lors de l'usage d'une implémentation, les causes d'éventuels problèmes.

3.3 Conception des suites de tests

3.3.1 Introduction et définitions

On appelle test ("test case"), ou aussi séquence de tests ou cas de tests, un ensemble d'interaction nécessaires pour vérifier une certaine propriété de l'IST.
On appelle suite de tests un ensemble de tests conçus pour vérifier la conformité d'une IST par rapport à sa spécification du protocole. Une suite de tests est composée de plusieurs groupes de tests qui à leur tour, sont composés de tests individuels. Chaque test individuel a son propre objectif de test, qui doit être suffisamment précis de façon à ce que le sens de l'échec soit clair. Les interaction qui composent un test sont aussi appelés événements.

3.3.2 Sélection de tests

Sélectionner des tests consiste à dériver des séquences d'interactions qui forment les tests, à partir d'une spécification de référence. C'est un domaine de recherche très actif, voir entre autres [Sabn 85], [Aho 88], [DeMe 91] [Dss 99].
On s'intéresse à dériver des cas de tests qui couvrent le plus grand nombre d'erreurs et de façon optimale. Une suite de tests de conformité, conçue pour un protocole particulier, doit être capable de tester tous les aspects obligatoires et optionnels sur une grande plage de variation des paramètres.
La conception d'une suite de tests doit tenir compte d'un certains nombre de facteurs et doit faire leur combinaison optimale. Les facteurs à considérer dans le cas des suites de tests pour un protocole à une seule couche sont les suivants:

1. **Les caractéristiques**: Une suite de tests doit tester toutes les caractéristiques optionnelles et obligatoires d'un protocole dans toutes les combinaisons possibles permises par les règles de conformité statiques de la spécification du protocole. Si la spécification du protocole permet un sous ensemble partiel de caractéristiques, comme par exemple "réception seulement", alors la suite de tests doit être adaptée à cette situation.

2. **Les phases du protocole**: Une suite de teste doit comporter des tests spécifiques à chaque phase fonctionnelle du protocole:
    - Établissement d'une connexion,
    - transfert de données,
    - fermeture d'une connexion.

3. **Les variations**: Une suite de tests doit inclure les variations suivantes:
    - variations de séquencement,
    - variations d'horloge et de synchronisation,
    - variations dans le codage des unités de données du protocole,
    - variations des paramètres dans les unités de données du protocole.

D'autres variations peuvent être considérées, comme les variations extrêmes et les valeurs erronées de paramètres.

4. **Comportement valide/invalide**: Une suite de tests vérifie le comportement valide de l'IST, mais aussi son comportement dans un environnement hostile. Pour cela les tests doivent comporter des branches qui décrivent un comportement invalide pour examiner la réaction de l'IST vis à vis de tels comportements.

5. **Interdépendance**: Une suite de tests doit tenir compte de l'interdépendance entre les différents sous-groupes de tests. Les tests de transfert de données, par exemple, ne peuvent être exécutés que lorsque les tests d'établissement de connexion et de transition vers l'état de transfert de données ont été exécutés avec succès. Une autre forme d'interdépendance est l'interdépendance fonctionnelle. Avant d'exécuter des tests de transfert de données, la capacité de négociation des longueurs des données doit être testée.

3.3.3 Sélection des tests fondée sur les automates à états finis

La sélection des tests dépend beaucoup du langage de spécification utilisé. Ainsi, les techniques de sélection associées aux langages basés sur les automates à états finis sont différentes de celles associées au langage LOTOS par exemple.

3.3.3.1 Le Test du flux de contrôle

Les méthodes de sélection de tests basées sur le flux de contrôle tiennent compte principalement des changements d'états, des entrées et des sorties. Les principales méthodes sont le tour de transition [Nait 81], UIO [Sabn 88], UIOv [Vuon 89], la séquence de distinction [Gone 70], W et Wp [Fuji 91].

3.3.3.2 Le Test des flux de données

Le test des flux de données est généralement basé sur les graphes de flux de données. Ces graphes sont des graphes dirigés dans lesquels les nœuds représentent des unités fonctionnelles d'un programme et les arcs le flux de données. Une unité fonctionnelle peut être une transition, une procédure ou même un programme.
Le test de toutes les valeurs d'entrées d'un programme donnerait une vue complète et globale du comportement de celui-ci, cependant l'espace des entrées se trouve être un espace très large, il est impossible de représenter toutes les données possibles. La procédure habituelle est de sélectionner un sous-ensemble représentatif de cet espace.

3.3.3.3 Quelques travaux et méthodes de sélection de tests de flux de données

Sarikaya [Sari 87] emploie des méthodes de test de programmes pour tester les flux de données. Il prétend que cette façon de tester procure les meilleurs résultats pour la détection des erreurs. Cette méthode utilise des graphes de flux de données pour modéliser le flux d'information d'une spécification ESTELLE en forme normale, excluant les changements majeurs d'états. Elle emploie des méthodes de décomposition et des méthodes de partitionnement fonctionnelles pour dériver des blocks de graphes de flux de données qui sont considérés comme des fonctions de flux de données. Ces fonctions sont alors testées à l'aide de paramètres de variations simulant tous les flux de contrôle

existant dans la spécification. Les dépendances de flux de données sont obtenues simultanément.

Dans [Ural 91], les auteurs appliquent les techniques de génie logiciels au test de conformité des protocoles. La technique utilisée est applicable à une spécification de protocole données sous forme d'une spécification ESTELLE en forme normale. En premier lieu, la spécification est transformée en un graphe de flux, ensuite chaque définition et usage d'une variable de contexte et chaque interaction de paramètres d'entrée/sortie employées dans la spécification sont identifiées. En se basant sur ces informations, les associations entre les sorties et leurs entrées qui peuvent influencer ces sorties sont établies, enfin des séquences de tests sont sélectionnées pour couvrir en moins une fois chaque association.

Miller [Mill 92] utilise une version limitée d'ESTELLE. Il transforme l'automate à états finis étendu (AEFE) en un AEF avec des modifications dans les entrées et les sorties. Le nombre d'états reste inchangé, par contre le nombre de transitions augmente. Un graphe de flux de données est construit à partir du AEF, des chemins de tests et des séquences de tests pour tester les flux de données et de contrôle sont générés à partir des graphes de flux de contrôle et de données, et combinés pour générer des séquences de tests exécutables.

Dans [Chan 93] tous les chemins "définition-usage" ([Weyu 85]) sont utilisés pour générer des chemins dont la première et la dernière transition sont dépendante soit par le contrôle soit par les données. Cette méthode utilise les techniques d'analyse de flux de données, tous les chemins "définition-usage" sont utilisés pour trouver les dépendances de contrôle et de données à travers les transitions du AEFE. Tester une spécification reviendrait donc à tester ces dépendances. Pour le test des flux de données, il suffirait de déterminer et tester les relations à travers les variables de la spécification du protocole.

Bourhfir [Bour 97, 99] a proposé une méthode automatique de génération de cas de tests pour les systèmes modélisés par des AEFEs. La méthode proposée est une méthode

unifiée pour le test du flux de contrôle et le flux de données. Elle génère des cas de tests exécutables pour tout système modélisé par un AEFE. Aussi, elle génère un plus grand nombre de cas de tests que les autres méthodes étant donné qu'elle utilise l'analyse de cycle qui consiste à insérer le cycle influent un certain nombre de fois dans un chemin non exécutable afin de le rendre exécutable. Cette méthode permet aussi de générer des tests pour des systèmes contenant des boucles non bornées.

4 Le diagnostic des protocoles de communication

4.1 Les spécifications de protocoles

Ce domaine a été le sujet de plusieurs travaux. Une bonne introduction peut être trouvée dans [Holz 91, Holz93]. Nous introduisons l'aspect spécification car nous devons définir le formalisme de spécification de protocole le plus approprié pour les techniques que nous proposons, nous donnerons par la suite les raisons de notre choix.

- **Les automates à états finis (AEFs) et les spécifications de protocoles**

Une référence de base sur la théorie des automates à états finis se trouve dans [Koha 78]. Une étude sur les formalismes de spécification de protocoles et leur pouvoir d'expressivité a été réalisée dans [Schw 82]. Elle inclut une catégorisation des formalismes suivant leur degré d'orientation vers les transitions d'états ou vers une exécution globale des séquences d'états. Une analyse des formalismes basés sur les AEFs pour le problème des protocoles pour le test de conformité est donnée dans [Barr 91].

- **L'automate à états finis (AEF)**

Un automate à états finis peut être représenté par un quntuple (S, I, Y, T, O) où :

S est l'ensemble des états incluant l'état initial S0

I est l'ensemble des symboles d'entrée

Y est l'ensemble des symboles de sortie incluant la sortie vide "-"

T est la fonction de transition d'états, S x I -> S

O est la fonction de sortie, S x I -> Y

La figure 2 donne un exemple de représentation d'un AEF à trois états et neuf transitions.

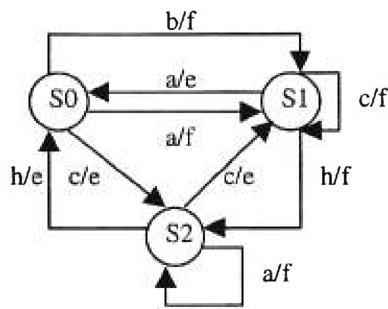


Figure 2 : Exemple d'un automate à états finis

Les automates à états finis (AEFs), sont utilisés sous différentes formes pour modéliser une large variété de systèmes dynamiques. Ils sont souvent utilisés pour modéliser les spécifications de protocoles, cependant les simples AEFs ont un pouvoir d'expressivité très limité, même pour modéliser de simples protocoles, parce qu'ils ne peuvent pas représenter des notions pratiques comme les "timers", les variables et les conditions de garde appliquées aux transitions. Les AEFs ont été utilisés pour le test des protocoles et les analyses de spécifications respectivement par [Wang 92]et [Lin 91]. Trois systèmes de diagnostic de protocoles ont été proposés utilisant des techniques basées sur AEFs [Boul 90, Ghed 93b, Ries 93a]. Tous ces travaux ont montré le besoin d'utilisation d'une modélisation plus riche de la spécifications du protocole. l'information disponible dans une spécification par AEFE est particulièrement importante pour la représentation de caractéristiques qui peuvent contenir des erreurs, et pour pouvoir réduire le grand nombre d'observations possibles.

Le fait que les simples AEFs et réseaux de pétri soient limités pour représenter les spécifications de protocoles est bien établi dans la littérature. Plusieurs propositions d'extension ont été faites et analysées, les plus importantes sont [Boch 77, Holz 91, Lin 91, Wang 92].

La limitation des AEFs pour le diagnostic des protocoles a été particulièrement reconnue dans [Boul 90] dans [Ghed 93a] et dans [Ries 93a].

- **L'automate à états finis étendu (AEFE)**

Un automate à états finis étendu est essentiellement un automate simple auquel des variables d'états sont rajoutées. Ces variables peuvent être munies de valeurs, ou peuvent être utilisées dans des expressions. Un prédicat opérant sur ces variables est associé à chaque transition et définit alors une condition qui doit être vérifiée avant l'exécution de cette transition. Un AEFE peut donc être vu comme un AEF dont les états sont définis par des conditions associées à leurs transitions. Une représentation d'un AEFE est donnée par la figure 3.

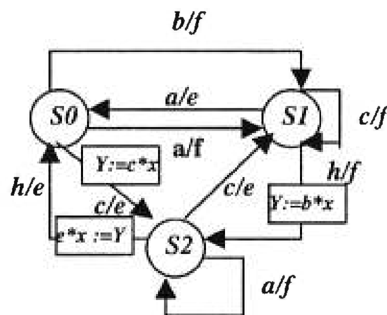


Figure 3 : Exemple d'automate à états finis étendu

Un AEFE est formellement représenté par un 7-tuple <Q, Σ, V, P, A, F , q0, > où :

Q est un ensemble fini d'états,

Σ est l'ensemble de messages qui peuvent être reçus et envoyés,

V est l'ensemble de variables de contexte,

P est l'ensemble des prédicats qui opèrent sur les variables,

A est l'ensemble des actions associées aux transitions

F est la fonction de transition qui associe Q x Σ x P(V) à Q x Σ x A(V),

q0 est l'état initial.

Chaque transition contient deux parties:

1- Une partie condition: Qui contient un événement entrée, un message, ou un prédicat représentant une fonction booléenne qui opère sur les variables de contexte et les paramètres du message d'entrée.

2- Une partie action: qui peut contenir des événements de sortie et des formules booléennes modifiant l'état des variables de contexte.

La transition ne peut être exécutée que si la partie condition est satisfaite. Quand la transition est réalisée, la partie action est exécutée et l'état du AEFE transfert de l'état tête à l'état queue de la transition.

Le AEFE est un modèle bien connu, non seulement pour les protocoles de communication mais aussi, pour tous les systèmes logiciels en général, il est une extension du AEF qui décrit uniquement la partie contrôle du système. L'extension est en fait l'addition de mécanismes qui permettent de décrire aussi bien les flux de données que le côté contrôle et acheminement de ces données.

4.2 Les approches de diagnostic pour les protocoles de communication

Les outils de diagnostic des protocoles peuvent être conçus pour plusieurs buts distincts. Le but peut être de rétablir le fonctionnement d'une implémentation opérationnelle, il peut être un test de conformité : tester la conformité de l'implémentation par rapport à sa spécification, il peut-être un test d'inter-opérabilité : tester si une ou plusieurs IST peuvent fonctionner ensembles et satisfont respectivement leurs buts spécifiés.
Dans ce travail, nous nous intéressons au diagnostic dans le cadre du test de conformité.

4.2.1 Les approches "système expert"

Il y' a quelques années, la plupart des approches automatisées de diagnostic des réseaux se restreignaient à l'utilisation de la première génération de systèmes experts décrits dans [Shap 92, Math 87]. Ces outils proposent des diagnostics en associant les informations alarmes et les observations du réseau aux informations de fautes collectées par expérience. Pour simplifier on appellera ces systèmes : systèmes experts.

Les systèmes experts sont très efficaces lorsqu'il s'agit de problèmes de routine, et quand les configurations et les composants du réseau ne changent pas et ne sont pas mis à jour. L'information d'un système expert collectée par expérience est souvent spécifique à une configuration ou implémentation de réseau particulière. Un tel système expert ne peut diagnostiquer de nouvelles fautes. La mise à jour du système d'une manière constante et fiable est une tache très difficile, les règles d'un système expert combinent souvent les mécanismes de diagnostic aux informations concernant des cas spécifiques et des détails d'implémentation bien spécifiques aussi.

Mis à part les systèmes experts, il existe des techniques qu'on appelle techniques de raisonnement par cas ou techniques de reconnaissance de modèle. Ces techniques tentent de déterminer si une erreur donnée est une variation d'un cas bien connu [Shaw 92, Lewi 93]. Malgré que ces approches soient flexibles, elles souffrent de l'incapacité de traiter de nouvelles fautes qui ne correspondent à aucun cas connu ou déjà vu.

Les systèmes experts et les techniques de raisonnement par cas fournissent des explications très limitées pour le diagnostic. Les explications se limitent typiquement au chaînage avant ou arrière utilisé ou aux règles qui relient les scénarios aux cas connus.

4.2.2 Les approches classiques

Quelques approches typiquement utilisées dans d'autres applications peuvent s'avérer efficaces pour le diagnostic des réseaux. Elles incluent 'les diagnostics', les dictionnaires de fautes, les arbres de décisions, et le diagnostic de systèmes à niveau.

'Les diagnostics' sont un ensemble de tests qu'on applique à une implémentation sous test afin de déterminer si elle fonctionne correctement et si certaines fautes ne sont pas

présentes dans l'implémentation. Ces tests peuvent être réalisés durant le développement du système ou après que celui-ci ait été trouvé non opérationnel. Cependant ce diagnostic ainsi défini est différent du diagnostic proprement dit, parce qu'il permet juste de vérifier si des comportements correctes sont présents et certains comportements incorrectes sont absents.

Les dictionnaires de fautes sont les parents de la première génération de systèmes experts, ils constituent une liste d'association entre les manifestations et les fautes.

Les arbres de décisions explorent systématiquement un arbre de nœuds de tests. La recherche du nœud de bas niveau est effectuée comme une fonction de décision faite au niveau supérieur. Par exemple pour diagnostiquer une automobile, le nœud supérieur teste si le circuit électrique est correct et celui du niveau inférieur test si l'accélération fonctionne.

Le diagnostic de systèmes à niveau, aussi connu sous le nom de diagnostic de fautes par les PMC [Prep 67], interprète les résultats de tests dans le but de localiser les fautes. Le test est une application de séquences d'entrées puis une observation d'une ou plusieurs sorties pendant un ou plusieurs intervalles de temps [Kime 86]. Ces systèmes supposent que les composants d'un système, par exemple un microprocesseur d'un système distribué, peuvent se tester eux-mêmes ou tester d'autres voisins. Ils sont appliqués aux circuits électroniques, bien que des essais avaient été faits pour les appliquer aux réseaux de communication [Kuhl 80]. Kuhl a supposé que les nœuds d'un réseau puissent se tester entre eux, comme dans le cas des systèmes multiprocesseurs. La faisabilité pratique de ces systèmes n'a pas été discutée.

4.2.3 Les approches de diagnostic fondé sur un modèle

Une bonne introduction au diagnostic fondé sur un modèle et comment il est comparé aux autres approches peut-être trouvé dans [Davi 88]. Les techniques de diagnostic fondé sur modèle comparent les observations du système en cours de diagnostic à celles attendues du modèle du système afin de diagnostiquer les fautes. Une erreur est définie comme un conflit entre les observations de l'implémentation et celles attendues du modèle. Ces systèmes ne dépendent pas des informations cumulées par expérience sur les fautes,

cependant cette information peut être utilisée si elle est disponible. Les nouvelles fautes peuvent être diagnostiquées si :

1- Leur manifestation peuvent être observées et ces observations sont représentées en quelques sortes dans le modèle,
2- leur comportement dans le système à diagnostiquer est représenté en fonction des relation entre les composantes du modèle.

Ces techniques fondées sur un modèle permettent des explications détaillées basées sur les principes importants du modèle, elles séparent le mécanisme du diagnostic des informations concernant le système sous diagnostic, elles n'essayent en aucun cas d'imiter l'expertise ou le raisonnement humain.

4.2.3.1 Avantages et limites des approches fondées sur un modèle

Les approches de diagnostic fondées sur modèle essayent de diagnostiquer les protocoles en analysant les conflits entre les observations et le modèle du protocole : la spécification. L'intérêt des approches basées sur un modèle pour le diagnostic des protocoles est motivé par le besoin d'une grande flexibilité, généralité, fiabilité et une grande capacité d'explication non offerte par les autres approches. Du fait que ces approches étaient basées sur un modèle, une erreur est définie comme un conflit entre les observations et les sorties attendues du modèle, ceci fait qu'elles ne nécessite pas une connaissance des fautes tirée des expériences précédantes, comme pour les systèmes experts.

Ces approches sont flexibles dans le sens où l'outil basé sur ces techniques peut-être appliqué à n'importe quel système qui utilise le même formalisme de modélisation. Par exemple n'importe quel protocole représenté par un automate à états finis. Ceci est du au fait que le mécanisme de diagnostic est complètement séparé du modèle de l'IST.

Elles sont fiables car le mécanisme de diagnostic ne change pas après son application au système et aussi parce que la notion de faute est basée sur l'observation du comportement

de l'IST par rapport au modèle correct, plutôt que sur des règles qui peuvent devenir inconsistantes ou inapplicables après transformation de l'IST.

Ces approches offrent une grande capacité d'explication parce que :

- le diagnostic indique exactement quels sont les composants correctes et ceux qui sont incorrectes,
- le diagnostic peut donner des hypothèses expliquant comment les composants se comportent anormalement,
- tous les diagnostics émis peuvent être justifiés en se rapportant au modèle, par exemple, en donnant la séquence de transitions expliquant les observations.

Ces approches sont pratiques. Elles permettent une interaction flexible avec l'opérateur pour construire un diagnostic. Par exemple le système de diagnostic peut donner des résultats intermédiaires des interprétations d'observations et suggérer des candidats de diagnostic qui peuvent être traités ensuite par l'opérateur.

Comme nous l'avons déjà souligné, les nouvelles fautes peuvent être détectées par les techniques basées sur un modèle. Le diagnostic des nouvelles fautes a été largement confirmé dans la littérature [Hams 92, Besc 93]. Un argument informel serait le vaste nombre de diagnostics possibles généré quand les fautes multiples sont considérées. Un opérateur ne pourrait considérer toutes les combinaisons des simples, multiples triples fautes et plus.

Les systèmes de diagnostic fondé sur un modèle représentent d'importants avantages, cependant ils ne sont pas parfaits, ils ne peuvent pas diagnostiquer des fautes qui ne sont pas observables, ils peuvent détecter, mais ne peuvent pas identifier exactement les fautes dont les manifestations sont insuffisantes pour permettre la discrimination par rapport aux autres fautes. Bien sur, cette limite est partagée par les autres techniques de diagnostic. Ces systèmes ne peuvent diagnostiquer que les fautes qui peuvent être exprimées dans le formalisme de modélisation utilisé. Par exemple, un formalisme pour le représentation de protocoles serait limité s'il s'agit de diagnostiquer des fautes de topologie. On pourrait penser que les systèmes experts seraient plus avantageux dans ce cas, parce que les règles

sont construites à partir d'heuristiques et ne sont associées à aucun modèle. Le plus grand désavantages des approches basées sur un modèle par rapport aux systèmes experts serait peut-être le fait que très peu d'outils aient été construits pour l'aide à leur développement. Cependant pour l'application que nous considérons, la spécification de protocoles est un domaine très matures où un nombre croissant d'outils de génération de modèles formels et bien consistants sont apparus (voir [Holz 91].

4.3 Les travaux en diagnostic

Cette section discute brièvement l'état de l'art et les travaux réalisés dans le domaine. Ces travaux se divisent en deux groupes, selon qu'ils se définissent comme outil de diagnostic ou outil de test et de détection de fautes. Parmi ces derniers ne nous intéressons qu'à ceux qui ont un lien avec le diagnostic.

4.3.1 Diagnostic de protocoles

4.3.1.1 Travaux de Bouloutas

Une importante contribution des travaux de Bouloutas a été un algorithme pour le diagnostic des AEFs simples, bien que son approche ne soit pas conforme aux systèmes conventionnels de diagnostic fondé sur un modèle, Bouloutas [Boul 90] la décrit comme étant fondée sur un modèle. Son algorithme de diagnostic considère le cas où les observations peuvent être corrompues et incomplètes, le modèle de la spécification est disponible et les modèles de fautes sont complets et disponibles.

Le modèle de la spécification est un automate déterministe dans lequel on ne fait aucune distinction entre les entrées et les sorties, on est dans le cas d'un réseau opérationnel. L'algorithme suppose que les observations de l'IST commencent à partir de l'état initial Q0. A partir des observations, les séquences de transitions fautives sont détectées, les fautes et leurs longueurs sont supposées connues, l'algorithme génère un ensemble L = { M, M'1, M'2,...} où chaque élément est appelé structure et doit inclure le modèle initial M et les modifications possibles qui permettront d'expliquer un peu plus les observations.

L'algorithme itere sur L en commençant par un singleton contenant M, une opération de modification est effectuée sur chaque élément de L de manière à expliquer les observations, la structure résultante est réinsérée dans la liste L. Bouloutas décrit l'algorithme comme une recherche en largeur, l'itération s'arrête après un nombre limité d'opérations effectuées sur chaque structure ou quand une des structure accepte toutes les observations. Le but est de trouver la première solution qui peut être considérée certaine. L'algorithme a été testé est implanté sur des petits ensembles d'AEFs. Cette approche n'a été testée sur aucun protocole de communication.

4.3.1.2 Travaux de Riese

La méthode proposée par Riese [Ries 93b] est similaire aux méthodes bien connues de diagnostic fondé sur un modèle. Les observations de l'IST sont propagées dans le modèle correct pour déterminer les conflits. L'algorithme de Riese, appelé HMDP (pour Heuristic Model-based Diagnostic of communication Protocols), modifie les parties du modèle de manière à réduire les conflits. Le modèle qui offre les meilleures explications des observations peut-être considéré comme la représentation de l'IST. Cet algorithme est perçu comme une recherche systématique de la meilleure explication des observations du système sous diagnostic.

Cette approche est particulièrement intéressante, elle apporte une nouvelle formulation du problème du diagnostic des systèmes modélisés par des AEFs et AEFs étendus. Le protocole est représenté de manière puissante par un ensemble de contraintes dérivées du AEF étendu, prenant en compte la richesse de l'information présente dans le modèle. L'ensemble des contraintes est alors utilisé pour la formulation du problème de satisfaction de contraintes (PSC), ouvrant ainsi le domaine à l'application des techniques de résolution existantes.

Bien que cette méthode se base sur la résolution de contraintes, les contraintes relatives au problème de diagnostic ont étaient définies, mais n'ont pas étaient concrètement utilisées pour réaliser le diagnostic. Une deuxième limite qui a été constatées, à mon sens, est que les contraintes du problème tendent à être de plus en plus importantes pour un modèle de protocole non trivial.

- Formulation du PSC

Le PSC est défini en terme d'un ensemble de variables V, un ensemble de domaines D des variables de V, un ensemble de contraintes C et un ou plusieurs buts ou solutions à atteindre. L'ensemble C est dérivé du modèle de manière à permettre la construction d'un réseau de contraintes [Shap 92]. Chaque variable Vi est associée à une observation Oi de l'IST. Le domaine Di de Vi est l'ensemble des transitions du modèle, chaque variable est représentée par un nœud dans le réseau de contraintes, les contraintes forment les arcs (une contrainte binaire forme un arc entre deux nœuds, tertiaire entre trois nœuds...), elles expriment le vocabulaire, le séquencement, les conditions de gardes, le timing, tout ce qui est relatif à une transition dans un AEFE.

- But de PSC

La solution est l'affectation, satisfaisant C, d'une transition du protocole à chaque variable correspondante, présente dans les observations. Intuitivement le but du PSC est de trouver une solution ou plus telles que les observations peuvent être expliquées par une ou plusieurs séquences plausibles de transitions, en ce basant sur le modèle de la spécification. Ceci veut dire que C (le modèle) ou V(les observations) doivent être modifiés pour qu'une solution puisse être trouvée. On est dans le cas d'un PSC dynamique.

- L'algorithme HMDP

Les approches fondées sur un modèle peuvent diagnostiquer n'importe quelle faute dont le comportement est exprimé par une relation entre les composants du modèle. Dans le cas de cet algorithme le diagnostic est défini comme une affectation des modes CORRECT, FAULTY, ou UNKNOW à chaque composant de l'implémentation, les composants fautifs peuvent être des éléments des ensembles de transitions, des conditions de gardes, des actions associées aux transitions ou à l'ensemble des conditions temporelles. L'implémentation de HMDP ne prend en compte que les relations relatives aux transitions et aux timing, l'algorithme peut déterminer toutes les fautes reliées à leur comportement .

Quand l'implémentation est fautive, le modèle explique très faiblement les observations, ceci résulte en un conflit entre les observations et le modèle qui s'exprime par la non satisfaction des contraintes de C. Cette information résultant de la première itération de l'algorithme est utilisée pour produire les candidats plausibles du diagnostic et indiquer les modifications qu'il faut apporter au modèle pour améliorer l'explication des observations. Chaque candidat est utilisé pour créer une variante du modèle M'i et chaque itération conséquente i évalue le candidat i. L'ensemble INTERP(M'i) est alors créé à la ieme itération par propagation des observation sur le modèle M'i. Si les résultats donnent la meilleure explication des observations alors le candidat i représente un diagnostic plausible et indique le composant fautif de l'IST. L'algorithme peut simplement réitérer jusqu'à ce que chaque membre M'i soit évalué, ou bien donner les résultats de l'évaluation des candidats à l'opérateur qui peut décider d'accepter le diagnostic, modifier M'i ou permettre la réitération.

4.3.2 Tests de protocoles

Les tests de protocoles sont réalisés dans le but de détecter les fautes présentes dans une implémentation de protocole, nous nous intéressons aux méthodes réalisées dans le but du test de conformité et spécialement celles qui sont liées de près aux méthodes de diagnostic.

4.3.2.1 Vuong et Ko

Vuong et Ko proposent une approche pour la génération de séquences de tests pour les protocoles de communication. Cette approche traite le problème comme un problème de satisfaction de contrainte [Vuon 90], elle a été testée et implantée pour des cas simples. Les auteurs discutent la possibilité de l'utiliser pour analyser la trace des résultats de tests. Ce travail a été le premier à présenter la formulation satisfaction de contraintes pour les protocoles de communication, Vuong et Ko ont été les pionniers dans ce domaine.

L'approche se base sur un formalisme d'automates simples, fortement connectés, minimaux avec l'existence d'une fonction « reset ». Le modèle est un automate

déterministe; à partir d'un état donné et une entrée il transfert toujours vers un seul état. Le types de fautes considérés sont les erreurs de données et les erreurs de transfert.

- L'approche de génération de tests

La séquence de test $in_i/ou_i.....in_n/ou_n$ est divisée en sous séquences individuelles in_i/ou_i . Chaque sous séquence définit une contrainte sur deux variables q_i, q_{i+1} qui correspondent aux états de la transition $(q_i, in_i) \rightarrow (q_{i+1}, ou_i)$. Le but du problème de génération de tests est de trouver un ensemble de contraintes qui peuvent être satisfaites uniquement par l'automate M, et générer alors la suite de tests TS correspondante. ce problème diffère d'un PSC conventionnel où les contraintes sont connues et la recherche se fait sur le modèle qui satisfait ces contraintes. La suite de test initiale est une « Transition Tour » de M, et peut être une séquence générée par n'importe quelle méthode de test classique.

Il peut exister n automates acceptant la séquence de tests TS, parce qu'il existe n automates qui satisfont les contraintes établies. D'autres contraintes additionnelles doivent être établies pour différencier M des n-1 automates. Il est prouvé qu'il faut log(n) sous séquences additionnelles pour différencier M.

- L'approche analyse des résultats

Les auteurs ont discuté la possibilité d'appliquer leur approche pour l'analyse des résultats de tests. Ils appliquent une séquence de test initiale à l'IST, puis les observations sont utilisées pour former un ensemble de contraintes. Des contraintes additionnelles sont ensuite ajoutées pour discriminer entre les différents modèles M' qui acceptent la séquence de test TS. Les auteurs n'ont pas expliqué comment déterminer la différence entre le modèle de la spécification M et le modèle M' de l'IST. Une possibilité serait de comparer La séquence acceptée uniquement par M et celle acceptée par M' et en comparant les ensembles de contraintes correspondants.

La limite de cette approche est qu'elle se base sur un modèle d'automate simple, ce qui fait que la formulation proposée ne se prête pas aisément à un formalisme plus puissant, tel qu'est le cas pour un AEFE. Par exemple parmi les contraintes proposées, la première

indique que pour une entrée donnée, une transition doit toujours transférer vers un état unique. Ceci n'est pas le cas quand des conditions de garde sont associées aux transitions.

4.3.2.2 La méthode de Ghedamsi

Ghedamsi [Ghed 93a] propose une approche pour le diagnostic des machines à états finis, dans le cadre du test de conformité. Il utilise les résultats des méthodes de génération de tests pour définir la suite de tests appliquée à l'implémentation. Le but est de trouver un ensemble minimum et complet de solutions considérées plausibles. Le type de fautes est supposé connu et la séquence de tests est supposée inclure des tests qui détectent chaque fautes même quand les fautes multiples sont possibles.

Ce travail est intéressants dans la mesure où il décrit la méthode dans les termes conventionnels du diagnostic fondé sur un modèle. Les tests sont appliqués à partir de l'état initial q0. Tous les conflits sont immédiatement détectés par le test qui est supposé couvrir tous les conflits. Le conflit est du soit à une erreur de transfert soit à une erreur de sortie. L'exécution des tests se poursuit, une entrée à la fois, jusqu'à ce que toutes les suites de tests TC sont exécutées ou un état incorrecte est détecté. Dans le dernier cas, l'exécution des tests est stoppée parce qu'on suppose que les fautes ultérieures ne peuvent être atteintes, et qu'un autre test est capable d'atteindre ces fautes, ce qui représente une forte assertion pour la couverture des fautes.

Pour chaque conflit détecté, un ensemble de candidats contenant toutes les transitions du chemin de test est construit. Chaque candidat affecte le mode correct ou incorrect de fautes de transfert et de sortie aux transitions. L'algorithme vérifie que les candidats couvrent bien tous les conflits, des tests additionnels sont ensuite effectués. Pour chaque candidat i, le modèle M est modifié et la séquence entière de tests est appliquée au nouveau modèle M'. Si les tests passent alors le candidat i est un diagnostic plausible expliquant les observations. L'algorithme s'arrête quand tous les candidats ont été testés. Puisque la suite de tests est complète, la stratégie de l'algorithme peut-être décrite comme une recherche complète sur tous les chemins.

L'algorithme a été testé et implanté sur plusieurs AEFs et sur une version simplifiée de la Classe Transport 0.

Dans la conclusion de sa thèse[Ghed 93a] Ghedamsi déclare que le diagnostic des AEFs étendus est un problème difficile et important. Une publication plus accessible pourrait être trouvée dans [Ghed 93b].

4.3.2.3 La méthode de Wang et Liu

Wang et Liu [Wang 92] proposent une méthode de génération de tests de conformité pour la détection des fautes présentes dans les implémentations de protocoles, leur technique se base sur une sémantique axiomatique, i.e. une évaluation symbolique des pré-conditions et post-conditions reliées aux déclarations d'un programme. Pour cette approche les déclarations correspondent aux séquences de transitions.
Le modèle utilisé est un AEF étendu fortement connecté où les transitions peuvent être libellées par des :

- données d'entrées et de sortie, qui correspond à l'envoie et la réception de messages,
- actions d'assignation, l'affectation des valeurs aux variables,
-conditions de garde qui définissent le contexte d'application des transitions.

Le temps n'est pas représenté, la transition initiale est la transition qui mène vers l'état initial.
L'algorithme proposé génère des tests qui vérifient que chaque transition du modèle réalise ses actions et transferts vers l'état attendu. Le but est de détecter toutes les fautes simples. Si les fautes présentes dans l'IST produisent un comportement incorrect observable, alors les résultats des tests sont considérés certains, le comportement incorrect sera forcément détecté.
Les fautes sont décrites comme une transition qui ne réalise pas ses action et ne transfert pas vers l'état souhaité.

Pour évaluer les conditions de garde associées aux transitions, lors de l'apparition d'un conflit, l'algorithme reprend tout le chemin du conflit, de la transition initiale V_{ini} du modèle de la spécification M jusqu'à V_j, la transition finale dans le chemin de conflit. V_j prendra toutes les valeurs qui sont possibles à partir de cet état mis à part le prochain état

de V_{ji}, c'est ce qu'on appelle un mutant pour V_j,. Cette notion de mutation est comparable aux modèles de fautes du diagnostic basé sur un modèle.

L'algorithme génère un test TP choisi de telle manière à ne pas produire les observations qui sont produites par les mutants de V_j, il établit ensuite une séquence entre les mutants de V_j et la dernière transition dans le test TP. Pour vérifier que l'action qui a été observée est exécutée, le chemin TP est étendu par une recherche en largeur sur tous les mutants de V_j, jusqu'à ce que l'effet de l'action puisse être vérifié. Ceci suppose qu'une telle extension de TP existe.

Pour les actions, la vérification est effectuée en observant la variable x correspondante à l'action, dans les messages de sortie ou par l'observation des autres variables dont les valeurs dépendent de x. L'algorithme détecte seulement les assignations incorrectes des transitions dont les valeurs de variables peuvent être lues ou déduites des messages de sortie.

Ce travail nous intéresse parce qu'il se base sur un AEFE, il traite les conditions de garde et les actions des transitions de protocole. Aucune information concernant l'implantation de l'algorithme n'a été mentionnée.

5 choix du formalisme de représentation et définitions

5.1 Choix du formalisme de représentation

Le formalisme choisi pour l'approche de diagnostic présentée dans ce travail est un automate à états finis étendu (AEFE) déterministe. L 'extension de ce formalisme par rapport à un AEF simple permettra la représentation des conditions de garde, les actions et les variables appliquées aux transitions. Cette notion d'extension du formalisme de l'automate avec de telles caractéristiques n'est pas nouvelle, cependant le formalisme présenté convient parfaitement au diagnostic. Comparé aux formalismes utilisés dans la plupart des approches de diagnostic existantes, il permet une meilleure représentation de la riche information présente dans les spécifications de protocoles.

Pour décider quel formalisme adopter, l'idéal serait de comparer différents prototypes utilisant des formalismes différents, malheureusement la recherche dans ce domaine n'est pas assez avancée pour permettre une telle comparaison. La plupart des approches relatives à ce travail sont basées sur une certaine forme d'automates à états finis. Wang et Liu avaient donné des raisons explicites pour le choix de ce formalisme par rapport aux autres [Wang 92], ils déclarent que le formalisme d'AEFE, comparé à Estelle et LOTOS, offre une bonne simplicité, permet une implémentation plus facile et une meilleure applicabilité en pratique. Le choix de Bouloutas [Boul 90] était évident puisque ses travaux étaient reliés à ceux de Hart [Hart 87]. Vuong et Ko avaient choisi le formalisme d'automates parce que leur approche de génération automatique de contraintes dépendait de ce formalisme. Quant à Ghedamsi, le choix de son formalisme était naturel, son approche profitait des résultats des tests des protocoles basés sur des automates à états finis. Riese a opté pour le formalisme de AEFE pour plusieurs raisons énoncées dans sa thèse [Ries 93a] parmi elles, les constructions temporelles qui permettent l'estampillage des observations, et le fait que le formalisme facilite la construction de l'ensemble des contraintes.

5.2 Le modèle de représentation

Un AEFE est essentiellement un AEF auquel des variables d'états ont été rajoutées. Ces variables sont utilisées dans les expressions, et sont aussi utilisées dans les définitions où elles reçoivent des valeurs. Un prédicat opérant sur ces variables d'états peut être associé à chaque transition d'état et définit alors une condition, qui doit être satisfaite. Si la condition n'est pas satisfaite, la transition ne pourra pas être exécutée.

Un AEFE est donc un AEF dont les états sont définis par des valeurs de variables et par les conditions associées à leurs transitions.

Nous nous intéressons de près aux états ainsi définis du AEFE et nous supposons que le AEF à l'origine est correct et dépourvu d'erreurs de sortie et de transfert de transitions qui sont indépendant des données des variables et des conditions de garde des transitions.

Soit un AEF, représenté par un graphe G = (V, E), où l'ensemble des sommets V représente l'ensemble des états, et l'ensemble des arcs E représente les transitions, chaque arc est libellé par i/o (input/output).

Notre modèle AEFE pourra alors être représenté par un graphe d'AEFE qui sera l'extension du graphe G du AEF, par l'adjonction de trois fonctions f1, f2, f3 telles que :

$f1 : I \times X \rightarrow P,\ f2 : I \times X \rightarrow O,\ f3 : I \times X \rightarrow X$

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

f1: définit la vérification du prédicat par les paramètres d'entrées et les variables d'états. Pour les transition spontanées $I = \varnothing$, et pour les transitions sans prédicat $P = \{vrai\}$.

f2: décrit l'influence des paramètres d'entrée et des variables de contexte sur les paramètres de sortie.

f3: définit le changement des variables d'état après l'exécution d'une transition.

5.3 Définitions[Weyu 85]

Une variable x d'une transition est dite une définition si $x \in I$ (l'ensemble des entrées), ou $x \in X$ (l'ensemble des variables de contexte) et l'exécution de la transition lui assigne une valeur. Quand l'exécution de la transition nécessite une valeur de x appartenant à l'ensemble des entrées, l'ensemble des sorties ou l'ensemble des variables de contexte, qui est obtenue à partir de la mémoire, nous disons que x est un usage. Plus précisément, x est un usage de calcul si x apparaît dans les paramètres de sortie ou dans le membre droit d'une instruction d'affectation de la transition.

x est un usage de prédicat si x apparaît dans la partie condition de la transition. 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 est une définition pour laquelle il existe un usage global dans les autres transitions. Sinon c'est une définition locale.

Dans l'exemple de AEFE présenté en figure 4, la transition (5) contient une définition et deux usages du paramètre d'entrée "ISDU", et une définition de la variable d'état "olddata". La transition (7) contient des définitions et des usages de prédicat des variables "Num" et "counter", un usage de calcul de "number" et "counter", et un usage de prédicat de "number".

5.4 Modèle de fautes

Le modèle de fautes que nous adoptons s'inspire du modèle de fautes du génie logiciel défini dans [Boch 91] pour les systèmes modélisés par AEF. Il se présente comme un raffinement du modèle de fautes originel, les fautes prises en comptes dépendent principalement des données de l'AEFE.

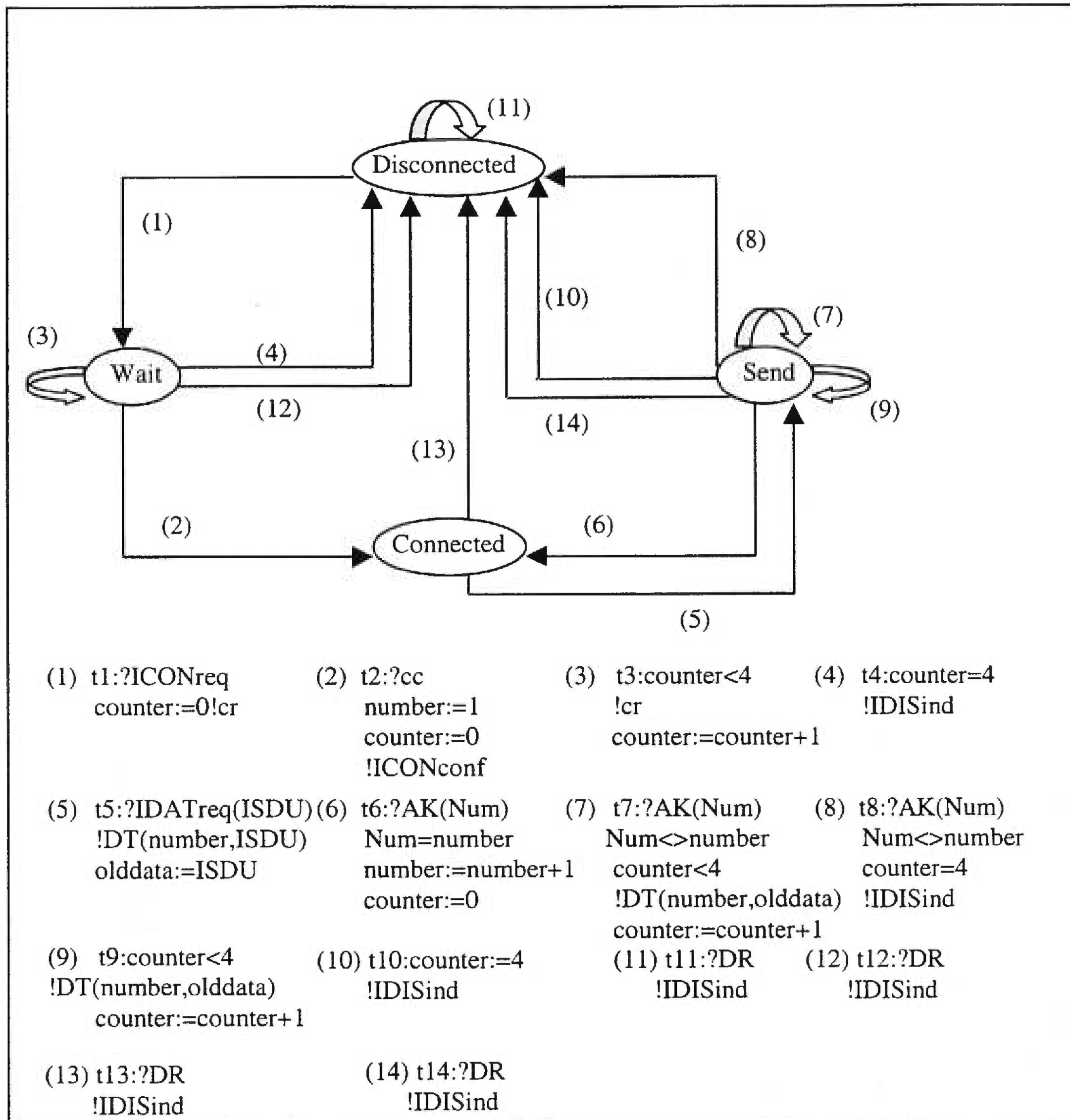


Figure 4 : Exemple de graphe d'AEFE.

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 rattachées reçoivent des valeurs différentes de celles définies dans le modèle du système. Ce qui se traduit 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. Il sera éventuellement le résultat d'une insatisfaction de la condition de la transition qui doit être exécutée ou de la satisfaction de la condition d'une

transition, autre que celle prévue par le modèle. De telles fautes sont dites des fautes latentes.

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 transfert vers un autre état autre que celui spécifié dans le modèle du système. Dans ce cas, cette faute de transfert est due au fait que la condition de garde de la transition ne peut être satisfaite. Dans ce cas on se trouve devant deux possibilités :

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

Pour ce type de fautes, 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 pour l'entrée correspondante, elle produit une sortie avec une valeur du paramètre autre que celui défini dans la sortie prévue du modèle.

6 Un algorithme général de diagnostic

Cet algorithme [Ghed 93a] concerne des systèmes structurés dont l'architecture est une composition de différents composants. Ce sont ces composants qui seront testés durant le processus de diagnostic pour déterminer s'ils sont correctes ou erronés. Nous sommes dans le cas où le comportement du système est décrit en terme d'entrées et de sorties, et que l'application d'entrées au système impliquera les composants du système et générera les sorties correspondantes.

Étant donné un système structuré et son modèle. le but de l'algorithme qui suit est de présenter les principales étapes requises par le processus de diagnostic. Un algorithme s'inspirant de cet algorithme général, et spécifique aux systèmes modélisés par un AEFE sera proposé et présenté dans la prochaine section.

Algorithme :

Étape 1 : Génération des sorties attendues

On suppose une suite de tests TS disponible, la suite de tests TS peut être obtenue par l'une des méthodes existantes de sélection de test. La suite de test est composées d'un ensemble des cas de tests, constitués de séquences de symboles d'entrées et qui permettent de passer par des chemins dans le système et du fait tester les composants sur ces chemins. Si on suppose que les cas de tests sont complets, nous serons alors de même à parcourir et tester toute l'implémentation.
La suite de tests TS est décrite par :

$TS = \{tc_1, tc_2, \ldots\ldots, tc_p\}$ où chaque tc_i est un cas de test.

Si le cas de test tc_i consiste en m_i entrées $i_{i,1}$, $i_{i,2}$,....., $i_{i,mi}$, la séquence des sorties attendues sera alors $o_i = o_{i,1}$, $o_{i,2}$,....., $o_{i,mi}$ où $o_{i,l}$ est la sortie attendue après l'application de l'entrée $i_{i,l}$ au modèle de l'implémentation.

Étape 2 : Exécution des cas de tests

Cette étape consiste en l'application de la suite de tests à l'implémentation sous test IST Pour chaque cas de test $tc_{i,}$, une séquence $\hat{o}_i$ de sorties observées est obtenue, où :

$$\hat{o}_i = \hat{o}_{i,1} , \hat{o}_{i,2} , ... \hat{o}_{i,mi}$$

Étape 3 : Génération des symptômes

Dans cette étape, on compare les sorties observées aux sorties attendues, et on identifie les différences. Chaque différence ($o_{i,j} \neq \hat{o}_{i,j}$) représente un symptôme.

Étape 4 : Génération des candidats potentiels

A partir des symptômes identifiés et du modèle du système, les candidats sont déduits. Chaque candidat diagnostic se définit comme la différence minimale, entre le modèle et son système, capable d'expliquer tous les symptômes. Cette différence indique une erreur dans un ou plusieurs composants du système.

Étape 5 : Discrimination entre les candidats

Le résultat de l'étape de génération des candidats potentiels se termine souvent par un grand nombre de candidats plausibles. Pour réduire cet espace de possibilités, deux techniques peuvent être utilisées. La première consiste en la génération de tests additionnels appelés "tests de distinction" (Distinguishing tests [Davi 84]). La deuxième technique consiste à augmenter les observations, au lieu de faire des tests additionnels.

7 L'algorithme et techniques proposés

L'algorithme qui sera décrit dans cette section se base sur certaines hypothèses concernant les fautes contenues dans l'IST et de la suite de tests appliquée pour détecter la présence de ces fautes. L'algorithme assure un diagnostic correct est complet si les hypothèses suivantes sont vérifiées:

Hypothèses:

pour chaque faute de l'IST, il existe un cas de test dans la suite de tests appliquée qui détecte cette faute,

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

Ceci se traduit par l'existence d'au moins un symptôme généré suite à l'application de la suite de tests. Chaque symptôme correspondra à une ou plusieurs fautes de l'IST.
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. Dans tous les cas nous supposons que la faute sera détectée.
On dit qu'une faute est directement atteinte par le cas de tests, si toutes les fautes antérieures à cette fautes ne sont pas des fautes de transfert.

L'algorithme proposé :

Étape 1 à 3 : Génération des sorties attendues, exécution des cas de tests et générations des symptômes
Ces trois étapes sont telles qu'elles ont été décrites dans l'algorithme général.

Étape 4 : Génération du graphe de dépendance des transitions

Dans un AEFE, les transitions se trouvent dépendantes les unes des autres. Cette dépendance est due aux variables de ces transitions. En effet une variable est définie dans une transition ti et utilisée comme variable de prédicat dans une ou plusieurs autres transitions tj, tk.

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

♦ **Procédure de génération du graphe de dépendance des transitions**

Cette procédure s'inspire de l'algorithme de [Chan 93], qui génère un graphe de dépendance de transition afin de déterminer les chemins exécutables d'un AEFE, dans le cadre d'une méthode de sélection de cas de tests pour les flots de données.

Cette procédure reçoit en entrée le modèle du AEFE en question, et fournit en sortie les dépendances définition-usage des variables x des transitions. On peut constater que si $x \in I$, alors x ne peut être qu'un usage Local, les usages postérieurs de x dans les autres transitions se feront via une définition globale de certaines variables d'états. Par exemple dans la transition (5) de la figure 4, "ISDU" est un usage local, son usage se fait dans la transition (9) à travers la définition de "olddata".

Maintenant que les paires définition-usage peuvent être aisément identifiables, nous définissons les dépendances entre les transitions, nous avons deux types de dépendances:

Dépendance des données : On dit que la transition ti est dépendante par ses données de la transition tj, s'il existe une variable x telle que :

- ti contient une définition globale de x,
- tj contient un usage de calcul de x (x est un paramètre de sortie ou x appartient à une action d'assignation),
- il existe un chemin "Def_Clear"2 de ti à tj.

Dépendance de contrôle : On dit que la transition ti est dépendante pour son contrôle de la transition tj, s'il existe une variable x telle que :

- ti contient une définition globale de x,
- tj contient un usage de prédicat de x (x est un paramètre de prédicat),
- il existe un chemin "Def_Clear"[3] de ti à tj.

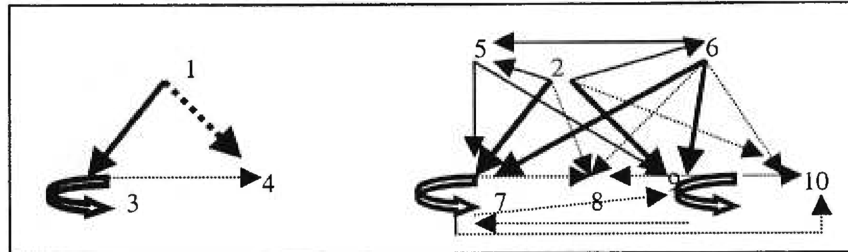


Figure 5 : Graphe de dépendances des transitions de l'exemple de la figure 4

Par exemple dans le modèle de la figure 4, 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.

Les flèches en trait plein représentent les dépendances de données, les flèches en pointillés représentent les dépendances de contrôle et les flèches en gras représentent les deux dépendances.

L'idée principale de la procédure (voir table 1) est de déterminer pour chaque usage de variable d'une transition du modèle, la ou les transitions qui contiennent les définitions des variables $x \in X$ exprimées par les usages de ces variables. Parce qu'une erreur de contrôle (usage de prédicat) ou de paramètre de sortie (usage de calcul) peut être due à une mauvaise définition des variables d'état sur lesquelles portent les usages. Ceci étant fait dans le but de proposer des hypothèses sur les transitions fautives.

---

[2] Un chemin de transitions (ti, t1.....,tk, tj) est appelé un chemin "Def-Clear" de ti à tj par rapport à une variable x, si le chemin t1,...tk ne contient pas de redéfinition de x. [Chan 93]

Dans ce contexte, la procédure que nous avons adaptée se définit comme réalisant le travail inverse que celui fait par Chanson [Chan 93] du fait que pour une définition donnée d'une variable, Chanson cherche le chemin dans lequel cette variable sera utilisée, soit pour un usage de prédicat soit pour un usage de calcul.

- **Étape 5: Construction des ensembles de candidats plausibles**

Nous construisons dans cette étape des ensembles de candidats. Un candidat est une transition notée par ^{o} , par * ou par ' , ce qui correspond, respectivement à une erreur de contrôle, une erreur de définition, ou une erreur de sortie. Par exemple pour l'ensemble {$t1^{o}$, t1*, t2*, t2' } représente la supposition que t1 a une erreur de contrôle et de définition, et t2 a une erreur de définition et une erreur de sortie.
Pour chaque cas de test donné tci, nous construisons un ensemble d'hypothèses à propos des transitions fautives.
Si on a m symptômes, on les considère dans l'ordre. Tous les symptômes postérieurs à une erreur de transfert (y compris les erreurs de définition, qui se manifestent par des erreurs de transfert) ne sont pas considérés, nous considérons qu'ils ne sont pas directement atteints par ce cas de test et qu'un autre cas de test pourra les atteindre directement.

- **Étape 6: Discrimination entre les candidats et génération des diagnostics (diagnoses)**

Un diagnostic est un candidat plausible dont l'affectation de fautes à ses transitions permettra d'expliquer les observations.
Pour discriminer entre les candidats, nous procéderons à une vérification qui consistera à appliquer les cas de test TS à la machine mutante, qui correspond au candidat en question. Si les sorties obtenues à partir de ce mutant sont identiques à celles observées sur l'IST, alors le candidat est confirmé.

Nous définissons un ensemble d'opérations, qui associe à chaque type de fautes un ensemble d'opérations de modification.

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

Pour les prédicats et les conditions associés aux transitions, l'ensemble des opérations à effectuer serait de remettre la condition à faux, ou de changer les bornes des valeurs du membre droit, quand celui ci est numérique (ex. counter < 4).

étape 1:

- Pour chaque transition, générer les ensembles :

  Def (t) = {x /x est une entrée ou x est une variable d'état définie dans le membre gauche d'une instruction d'action}
  C-Use (t) = {x / x est un parametre de sortie ou une variable d'état définie dans le membre droit d'une instruction d'action }
  P-Use (t) = {x / x est une variable de prédicat}

- Soit T(t, x) l'ensemble des transitions traversées à partir de t par rapport à la variable x.

  T(t, x) $\leftarrow \varnothing$ transition non encore visitée.

étape 2:

- Pour chaque transition t faire :

```
Si C-Use(t)  ou P-Use(t) est non vide
  Pour tout x ∈ C-Use (t) et x ∈ P-Use(t) faire:
    T(t, x) ← ∅
    search (t, x)
    marquer toutes les transitions non visitées.
```

étape 3:

- extraire les chemins définition-Usage

search (t, x)

```
        Marquer t par "visitée"
        Pour chaque transition entrante u à l'état de départ de t Faire:
        Si u est non visitée
                si x ∉ Def (u)
                  ajouter u àT(t, x)
                  sinon si  x ∈ Def(u) et x ∈ C-Use (t)
                      marquer t "data dépendante" de u
                      si  x ∈ Def(u) et x ∈ P-Use (t)
                       marquer t "contrôle dépendante" de u
                search(u, x)
Fin
```

Table 1: procédure de génération des dépendances des transitions

8 Conception du système de diagnostic: "Extended Multiple Fautes Diagnostic Tool (EMFDT) "

Nous donnons dans cette section une description du système qui a été réalisé. Ce système est un outil de diagnostic fondé sur un AEFE, il s'applique aussi bien au cas où le protocole sous diagnostic comporte une seule faute ou dans le cas où il comporte plusieurs fautes. Les fautes considérées sont des erreurs qui dépendent des variables du système. Des erreurs de contrôle quand il s'agit de fautes contenues dans les conditions des prédicats, des erreurs de définition lorsqu'il s'agit de fautes au niveau de l'action de la transition, et des erreurs dans les sorties quand il s'agit d'erreurs au niveau des paramètres de sorties de la transition.

Ce système permet le diagnostic des protocoles de communication modélisés par des AEFEs, il permet la production d'un ensemble minimal de diagnostics, qui expliqueraient les anomalies observées sur le comportement d'une implémentation sous test.

L'outil analyse la trace d'exécution d'une suite de test sur une implémentation, pour produire des hypothèses qui expliqueraient les erreurs détectées.

Pour assurer l'exactitude des résultats obtenus, nous appliquons la technique de mutation à l'ensemble des mutants.

Pour illustrer toute les étapes réalisées par l'outil, un exemple sera présenté en parallèle avec la description du système.

8.1 Architecture conceptuelle du système

Nous avons choisi de réaliser le système en adoptant une architecture modulaire. Chaque composant ou module réalise une tâche bien précise. Chaque module est muni d'une interface tout en restant ouvert à une communication avec les autres.

La figure 6 montre le premier niveau de décomposition de l'outil EMFDT. Cette version de l'outil traite des spécifications basées sur des modèles d'AEFEs et des traces d'exécution avec un modèle d'Entées/Sorties. On peut distinguer deux types d'ellipses sur la figure. Des ellipses pleines et vides, les ellipses vides désignent les modules qui

s'exécutent une seule fois durant le processus de diagnostic, alors que les ellipses pleines désignent les modules qui s'exécutent pour chaque cas de test.

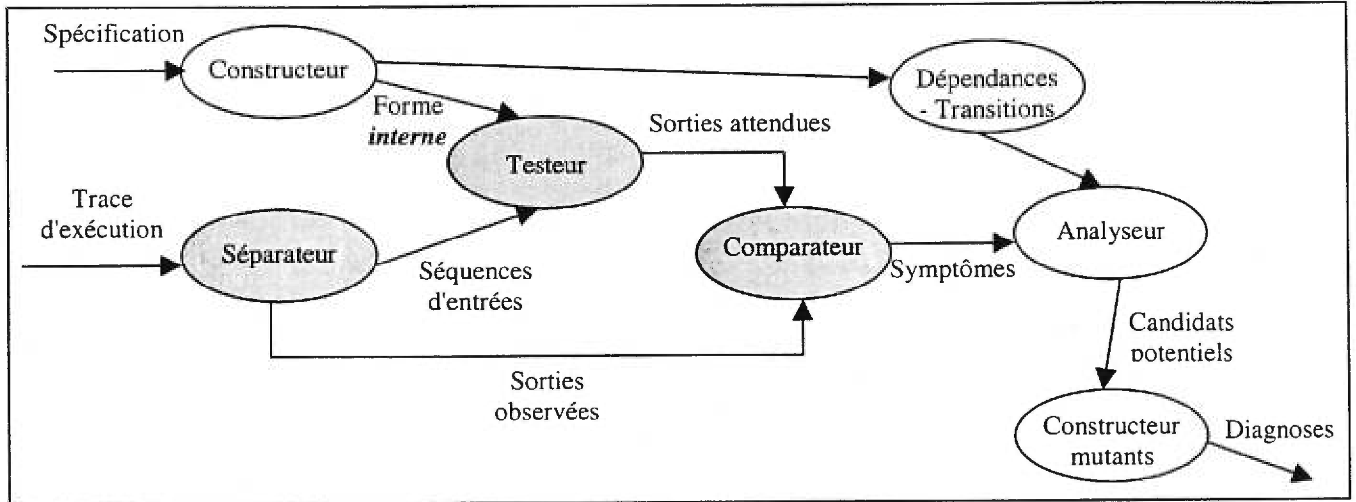


Figure 6 : Décomposition en modules de l'outil "EMFDT"

8.2 Exemple

Pour illustrer les différentes étapes de l'outil, nous considérons l'exemple de la figure 4 qui représente une spécification correcte d'un système et l'exemple de la figure 7 qui représente une implémentation erronée de cette spécification Dans ce qui suit, nous montrerons comment procède EMFDT pour localiser les erreurs.

La suite de test qu'on a appliquée à la spécification erronée (figure 7) de cet exemple est la suivante:

Cas de test # 1	r, ?ICONreq, ?-, ?- , ?cc, ?dr
Cas de test # 2	r, ? ICONreq, ?cc, ?IDATareq(ISDU=1), ?AK(NUM=3)

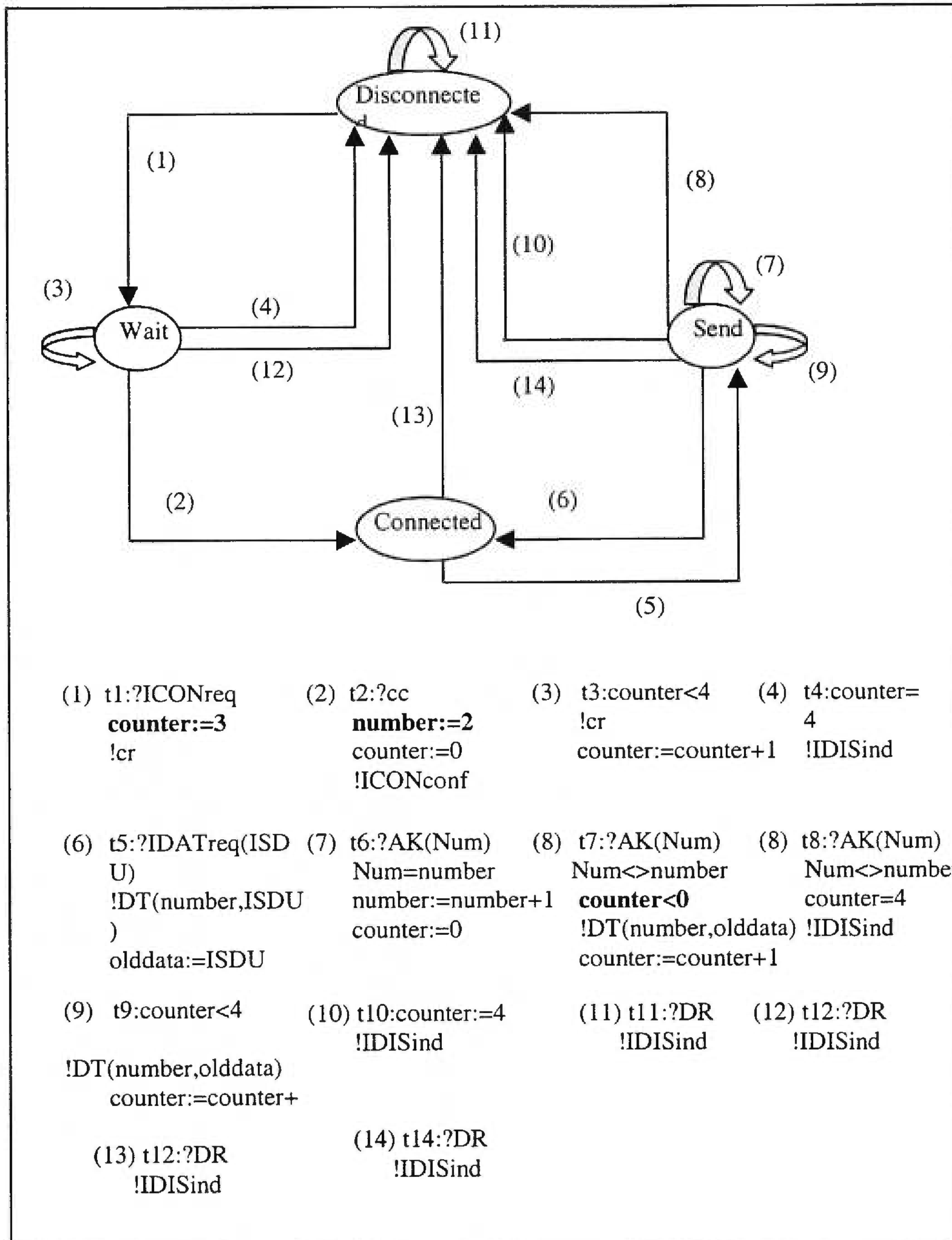


Figure 7 : Implémentation erronée de la spécification de la figure 4

(les erreurs sont en gras)

où "r" désigne un "Reset", une entrée qui ramène l'implémentation à l'état initial S0 à partir de n'importe quel autre état. Aucune sortie ne correspond à cette entrée, pour décrire ceci, nous avons utilisé le signe "-" pour la sortie. Ce signe est également utilisé pour les transitions spontanées (sans aucune entrées), et pour les transitions qui ne produisent aucune sortie.

- **"Constructeur"**: C'est le premier module qui reçoit la spécification du système sous diagnostic. La spécification est donnée sous forme d'un fichier texte organisé selon un format bien spécifique (voir ANNEXE A). le rôle du constructeur est de transformer cette spécification reçue en une forme interne, sans perte d'information. Comme nous avons opté pour une conception orientée objet, la forme interne n'est autre qu'un ensemble d'objets et de pointeurs qui lient les objets entre eux. l'utilisation d'objets se justifie par des besoins de réutilisation et de facilités d'extension, les pointeurs servent le besoin d'accès rapide lorsqu'on consulte les éléments de la spécification, chose qu'on fait fréquemment dans plusieurs modules (Tester, Analyseur, Dépendance-transition...). Le modèle objet de la forme interne exprime un modèle général et ouvert qui prend en compte tous les concepts nécessaires à la réalisation des objectifs visés par un tel outil. Ce modèle est facilement extensible et permettra la prise en compte des nouveaux aspects correspondant à de nouveaux objectifs. Le modèle objet du système est représenté par la figure 11 dans la section 10.3.

- **"Séparateur"**: Les traces des exécutions des suites de tests sont données selon un formalisme spécifique représenté par une suite d'entrées/sorties de la forme: r!-, ?ICONreq!cr, ?cc!ICONconf, ?IDATreq(ISDU=1)!DT(2, 1), ?AK(NUM=3)!IDISind, où les entrées sont précédées par "?", les sorties par le caractère"!". Le rôle du module "Séparateur" est de fournir les données nécessaires aux modules "Testeur" et "Comparateur". En effet, il extrait de la trace d'exécution, générée par l'application de la suite de tests sur l'implémentation, les entrées pour le "Testeur" et les sorties pour le "Comparateur". Pour extraire ces séquences d'entrées et de sorties de la trace d'exécution, le séparateur doit parcourir tous les enregistrements des résultats des

différents cas de tests. Pour l'exemple de la figure 4 , on aura les résultats de la figure9.

Cas de test # 1	r!-, ?ICONreq!cr, ?- !cr, ?-!IDISind , ?cc!-, ?dr!IDISind
Cas de test # 2	r!-, ?ICONreq!cr, ?cc!ICONconf, ?IDATreq(ISDU=1)!DT(2, 1), ?AK(NUM=3)!IDISind;

Figure 8 : Trace d'exécution (Comportement observé de l'IST)

- **"Testeur":** Les sorties observées sont directement extraites de la trace d'exécution. Pour avoir les sorties attendues, on doit appliquer à la spécification, plus précisément à sa forme interne, les séquences d'entrées de la suite de tests. Avant l'application d'une séquence de tests, on vérifie si la spécification la supporte (i..e. on vérifie si pas de réception spécifiée). Si la séquence n'est pas supportée, un message d'erreur avertie l'utilisateur en lui donnant l'identificateur du cas de tests qui sera ignoré dans la suite du processus. Les entrées de la séquence de tests sont appliquées une à une au modèle de la spécification. Pour chaque entrée de la séquence, le ``Testeur`` exécute les étapes suivantes :
  - recherche les transitions qui prennent l'entrée de la séquence de test,
  - recherche parmi les transitions trouvées celle qui sera exécutée; celle dont la condition sera vérifiée compte tenu des valeurs des variables d'état,
  - exécute la transition retenue, réalise ses actions et mets à jour les variables d'état.

Cas de test # 1	!-, !cr, !cr, !IDISind , !-, !IDISind
Cas de test # 2	!-, !cr, !ICONconf, !DT(2, 1), !IDISind;

Figure 9 : Sorties extraites

- "**Comparateur**": comme son nom l'indique, ce module compare les sorties observées extraites de la suite de tests aux sorties attendues, recueillies à la sortie du "Testeur". La comparaison se fait aussi bien pour les valeurs des sorties que pour les paramètres de ces sorties. Si les sorties sont identiques, les paramètres de sortie sont comparés et si tout est conforme, il confirme l'exactitude de la sortie de la première transition rencontrée. Sinon., il génère des symptômes qui seront analysés par la suite. Les symptômes générées pour l'exemple sont donnés dans la figure 10.

- "**Analyseur**"*:* ce module reçoit en entrée les symptômes du "Comparateur". Son rôle est de retrouver les causes qui expliqueraient les symptômes reçus. A partir du graphe de dépendances des transitions et des symptômes, l'analyseur génère un ensemble d'hypothèses sur les transitions suspectées fautives. Dans le cas d'une erreur de transfert, l'analyseur génère deux hypothèses. Une hypothèse pour la condition de garde de la transition, suspectant sa transcription erronée. L'autre pour les variables de la condition, les suspectant mal définies. Dans ce dernier cas l'analyseur doit remonter aux transitions qui définissent les variables et émettre des hypothèses a propos des définitions des variables de ces transitions. C'est dans ce cas, que sont utilisés les chemins ``définition-usage`` du module ``Dépendances-Transitions``. Dans le cas d'une erreur de paramètres de sortie, l'analyseur émets des hypothèses sur les transitions qui définissent les valeurs des paramètres. Les hypothèses de tous les cas de tests sont par la suite combinées entre elles afin d'avoir des hypothèses globales qui couvriraient toutes l'implémentation. Le résultat obtenu est une suite de classes de mutants parmi lesquels il existerait forcément un mutant qui représente l'implémentation erronée. Pour notre exemple nous aurons les résultats de la table 2.

Entrées	ICONrec	-	-	cc	dr	ICONrec	cc	idatarep(1)	AK(3)
Sorties attendues	cr	cr	cr	ICONconf	IDISind	cr	ICONconf	DT(1,1)	DT(1,1)
Sorties observées	cr	cr	IDISind	-	IDISind	cr	ICONconf	DT(2,1)	IDISind
Transitions (spec)	t1	t3	t3	t2	t13	t1	t2	t5	t7

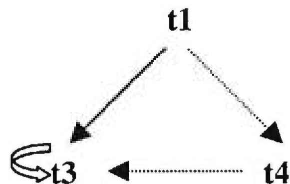
Symptômes

Figure 10 : Symptôme du cas de l'exemple 4

En effet:

- **Pour le cas de tests #1**

D'après le graphe de dépendances des transitions on a :



t3 est dépendante par les données et le contrôle de la transition t1, car t3 contient un usage de calcul : "counter:= counter + 1", et un usage de prédicat: "counter <4" de la variable "counter" définie dans t1. Une erreur dans la transition t3 pourrait être due à une erreur de définition de la variable "counter" dans t1. La transition t4 n'est pas prise en compte puisqu'elle n'est pas dans le chemin du cas de tests concerné.

		Candidats corrects	Candidats faux (variables)
Cas de test #1	Hypothèse #1	t1	t3^{o}
	Hypothèse #2	t3	t1*(counter)
Cas de test #2	Hypothèse #1 (symptôme 1)	t2	t7^{o}
	Hypothèse #2 (symptôme 1)	t7	t2*(counter, number)
	Hypothèse #3 (symptôme 2)	Aucune hypothèse	t2*(number)

Table 2 : Liste des hypothèses

Le symptôme au niveau de la transition t3 (figure 12) peut être expliqué par :

- H1: erreur de définition dans la transition t1 (t1*),
- H2: erreur de contrôle dans t3 (t3^{o}).

Les symptômes ultérieurs de ce cas de test ne seront pas considérés, ils sont postérieurs à une erreur de contrôle, ils ne sont pas directement atteints par ce cas. Un autre cas de tests pourra les atteindre

♦ **Pour le cas de tests #2** :

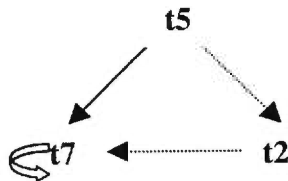
- **Premier symptôme : t5**

Le paramètre de sortie (number) a une valeur différente de celle du paramètre attendu, d'après le graphe des dépendances des transitions (figure7), on voit que la transition t5 est Dépendante de la transition t2 par rapport à la variable number. Ce sumptôme est du :

♦ H1: erreur de définition dans t2 (number = 1 au lieu de number = 2).

- **Deuxième symptôme : t7**

D'après le graphe de dépendances des transitions on en déduit que :



t7 est dépendante de t5 par les données au niveau de la variable, "olddata", définie dans t5 et est dépendante par les données et le contrôle de la transition t2 pour les variables "counter" et "number" définies dans t2.
Étant donné que "olddata" qui figure dans la liste des paramètres de sortie, n'est pas une variable du prédicat, une erreur au niveau de sa définition n'influerait en aucun cas sur le contrôle de t7, les variables à considérer sont plutôt "counter" et "number".

Le symptôme au niveau de t7 peut être du à:

♦ H1: erreur de transfert dans t7 et t2* correcte,
♦ H2: erreur de définition de counter ou number dans t2 et t7 ^{o} correcte.

Remarques :

- Pour les erreurs de transfert, les variables pertinentes sont les variables d'usage de prédicat. En d'autres termes les variables de prédicat.
- Pour les erreurs de sortie, les variables pertinentes sont les variables d'usage de calcul, les paramètres de sortie.

Après combinaison de toutes les hypothèses pour l'ensemble des cas de tests, nous obtenons les classes { t3°, t7 °}, { t3°, t2*(number,counter}, {t3 ° , t2* (number)}, { t1*(counter), t7 °}, { t1*(counter), , t2*(number,counter}, { t1*(counter), t2* (number)}. où juste les candidats faux sont représentés.
La combinaison se fait de façon à ne prendre qu'une seule hypothèse par cas de test.

- "**Constructeur de mutants**": Ce module offre un moyen de réduire l'ensemble des candidats potentiels générés par l' "Analyseur". En effet, on utilise dans cette étape la technique de mutation pour vérifier si le candidat réagit de la même manière que l'IST après l'application de la suite de tests. Par exemple pour la classe mutant { t3°, t7 °}, on commence par construire un mutant à partir de la spécification, en lui injectant une erreur de transfert dans t3 et t7.

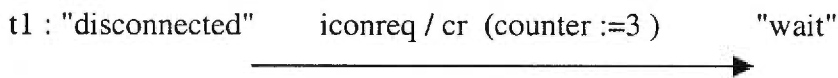
Le problème qui se pose est au niveau de la détermination des erreurs. Comment construire les erreurs à injecter et quelles valeurs donner aux variables d'états?
Une approche, pour cette version de l'outil, serait de construire les mutants en attribuant aux variables des instructions d'action (les variables dont on ne peut observer l'évolution) les valeurs limites des intervalles les définissant, ou même toutes les valeurs de l'intervalle si celui ci est réduit. Pour les autres variables, celles dont les valeurs apparaissent au niveau des paramètres de sortie des transitions, on fera une exécution symbolique sur l'IST pour déduire les valeurs des données qu'on suppose erronées.

La solution proposée n'a pas été implantée. C'est une solution temporaire car les valeurs des variables erronées ne sont pas forcément les valeurs limites des intervalles qui définissent les variables d'états, en plus ces variables peuvent être définies sur de larges intervalles. De nouvelles stratégies de sélection des valeurs des variables doivent être trouvées et implantées et ainsi enrichir l'outil.

Une fois ces erreurs déterminées, on les injecte dans le modèle correcte pour constituer le mutant, on observe alors son comportement. Si le mutant se comporte comme l'IST, il est accepté sinon il sera écarté.

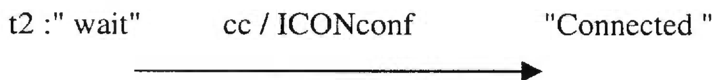
Pour cet exemple le mutant retenu correspond à la classe { t1*(counter), t7°}.

t1 : "disconnected" iconreq / cr (counter :=3) "wait"

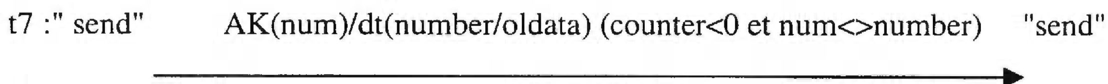


Erreur de définition dans t1. counter a été initialisé à 3 au lieu de 0.

t2 :" wait" cc / ICONconf "Connected "



t7 :" send" AK(num)/dt(number/oldata) (counter<0 et num<>number) "send"



Première expression de la condition a été altérée. counter < 0 au lieu de Counter < 4.

Ce mutant correspond exactement à l'IST (figure 7). En général on ne peut pas garantir un résultat final avec un seul mutant, le nombre d'éléments retenus dépend principalement de la couverture de la suite de tests.

9 Implantation du système EMFDT

9.1 Représentation interne du système

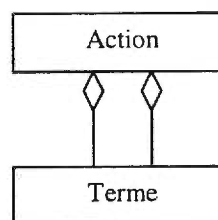
Tout au long du développement de notre système, nous avons continuellement considéré une éventualité d'extension de l'outil et de réutilisation de certaines de ses parties, nous avons donc opté pour une conception orientée objet en adoptant une méthode de conception claire et efficace OMT [Derr 95] [Rumb 91].

9.2 Représentation graphique des structures de données

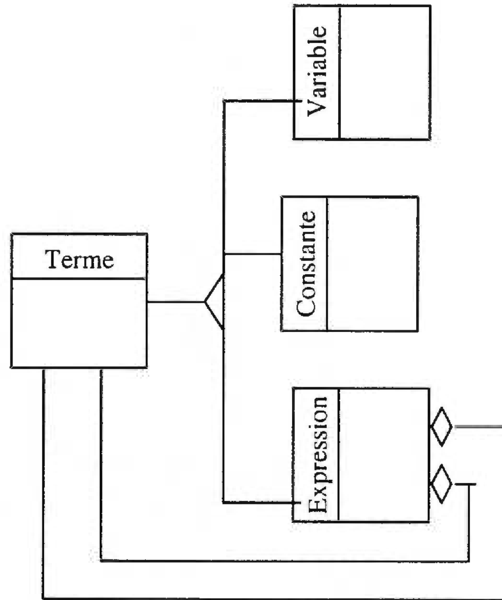
La représentation des données est effectuée à l'aide du formalisme objet avec des classes représentant des types et des liens entre ces classes. Une telle représentation a été choisie en raison de sa simplicité et parce que le C++ , le langage utilisé, est un langage orienté objet.

Nous avons utilisé la notation OMT ("Object Modeling Techniques") [Rumb 91]. Avec cette notation, chaque classe d'objets est représentée par un rectangle portant son nom. Les différents types de liens entre les classes sont:

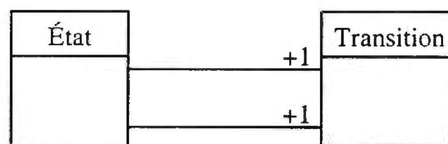
Composition simple: La présence du symbole "◇" entre une classe A et une classe B, signifie qu'un objet de la classe A est composé d'un seul objet de la classe B. Par exemple, un objet de la classe **Action** est composé d'un objet de la classe **Terme**, qui représente le terme gauche de l'action et d'un autre objet de la classe **Terme** qui représente le terme droit de l'action



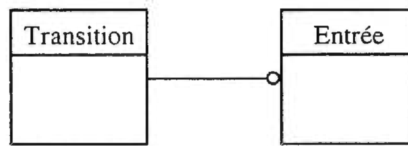
Lien d'héritage: La présence du symbole "<| " entre une classe A et une classe B, signifie que la classe B hérite de la classe A. Par exemple, les classes **Variable**, **Constante et Expression** héritent de la classe **Terme**, signifiant qu'un terme peut être une variable, une constante ou une expression. Comme dans le cas d'une action de transition où le terme droit peut appartenir à l'un des trois types de classes.



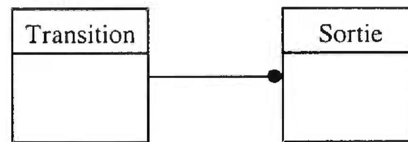
Lien simple: La présence du symbole " —— " entre une classe A et une classe B, signifie que la classe B est liée à une seule occurrence de la classe A. Par exemple, le lien entre État et Transition dans le modèle, une transition a un état de départ et un état final.



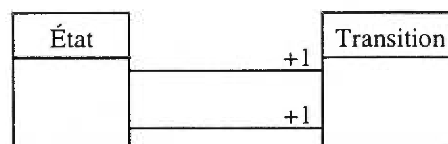
Lien au plus un: La présence du symbole "——O" entre une classe A et une classe B, signifie que la classe A est liée à zéro ou une occurrence de la classe B. Par exemple, le lien entre Transition et Entrée. Une transition contient au plus une entrée, ou pas d'entrée du tout dans le cas d'une transition spontanée.



Lien zéro ou plus : La présence du symbole " ——● " entre une classe A et une classe B, signifie que la classe A peut être liée à zéro ou plusieurs occurrences de la classe B. Par exemple, le lien entre Transition et Sortie. Une transition peut avoir plusieurs sorties, ou pas de sortie comme dans le cas d'une transition qui représente l'incrémentation d'un compteur de boucle.



Lien un ou plus : La présence du symbole " —+1—" entre une classe A et une classe B, signifie que la classe A peut être liée à une ou plusieurs occurrences de la classe B. Par exemple, le lien entre État et Transition. Un État appartient à une ou plusieurs transition.



9.3 Le modèle de données du système

Le modèle de données du système se définit principalement à l'aide de deux éléments, qui permettent la réalisation du diagnostic. Ces éléments sont le modèle de la spécification et la trace d'exécution de l'IST. Comme nous l'avons décrit dans la figure 5, la trace est formée d'un ensemble de cas de tests, et un test est composé d'une suite d'entrées et de sorties, exprimant les cas de tests réalisés sur l'IST et les sorties qui ont été observées.

Le modèle de la spécification, quant à lui se définit à partir d'un ensemble de transitions et d'états. Une transition à un état de départ et un état d'arrivée, elle contient des données représentées par les entrées, elle comporte des sorties, des conditions d'exécution de la transition, et l'action réalisée une fois que la transition est exécutée.

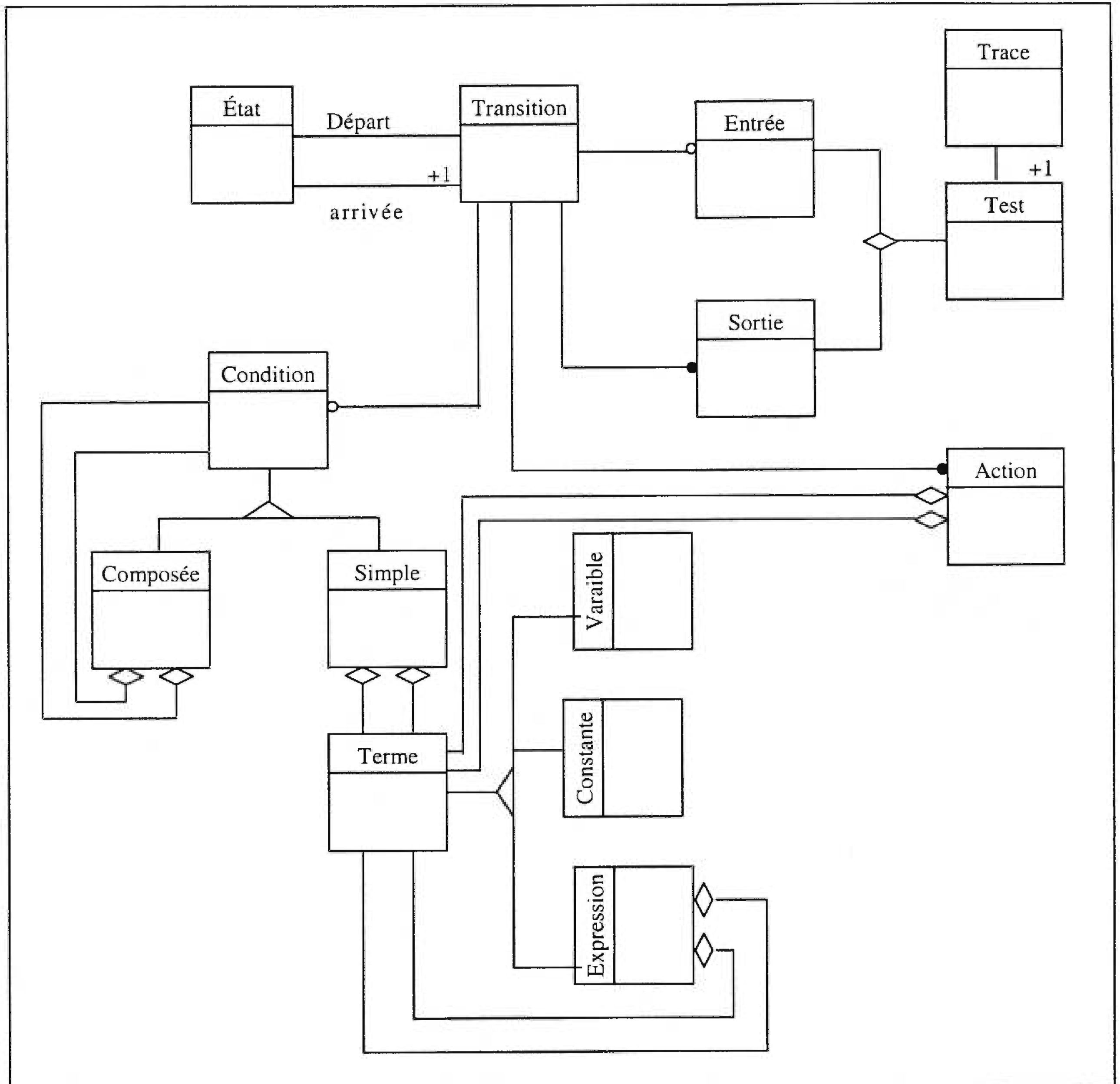


Figure 11 : Modèle de données du système

Les conditions associées aux transitions peuvent être simples ou composées. Les transitions simple sont de types "Counter < 0" , "Counter = Num"..etc. Chaque condition est formée d'un terme gauche, qui est en fait une variable et d'un terme droit qui peut être une variable ou une constante. Les conditions composées sont des conditions simples composées à l'aide d'opérateurs logiques de type , ET, OU....etc.

Les actions sont en fait des instructions de type "Counter:= Counter +1", "Number := 0", Number := Num....etc. Une action est composée de deux termes, un terme gauche qui est en fait une variable, et un terme droit qui peut être une variable, une constante ou une expression comme dans le cas de "Counter:= Counter +1".

Au début de ce projet, la spécification a été donnée sous forme d'un fichier texte écrit dans un format de représentation spécifique (voir ANNEXE A). Un grand effort d'abstraction a été nécessaire pour pouvoir dégager les concepts essentiels du modèle de la spécification, bien représenter les liens entre eux, et se focaliser sur l'aspect données afin de le représenter de manière claire et précise.

L'étape de la conception a été relativement longue, néanmoins elle a permis de maîtriser les concepts et aboutir à une implantation simple, rapide et efficace.

9.4 Le langage de développement

L'application a été développée avec Visual C++ ® 6.0. Les librairies de classes suivantes ont été utilisées :

- Rogue Wave Tools.h++ : Les principales caractéristiques de cette librairie sont comme suit :
  - Classes de base : ensembles, collections, chaînes de caractères, expressions régulières formats date et heure, listes chaînées.
  - Flux calibrés : permettent de manipuler un format binaire portable pour l'échange de données entre Unix Windows et OS/2.
  - Internationalisation des applications.
  - Support des bases ObjectStore : possibilité de placer des instances de classes Rogue Wave dans la mémoire persistante d'une base ObjectStore.
  - Réduction de la taille du code.

- Microsoft® Foundation Class (MFC) library permettant de créer des applications et des contrôles sur la plateforme Windows

Principales structures de données utilisées

- RWTValOrderedVector<T> : Classe RogueWave permettant de maintenir une collection de valeurs implantées sous forme de vecteur. Le paramètre T désigne le type de la valeur.

  Exemple : RWTValOrderedVector<int> est un vecteur ordonné de valeurs entières.

- RWTPtrOrderedVector<T> : Classe RogueWave permettant de maintenir une collection de pointeurs implantés sous forme de vecteur. Le paramètre T désigne le type pointé par chaque élément de ce vecteur.

  Exemples :

  1. RWTPtrOrderedVector<Sortie> est un vecteur dont chaque élément est un pointeur vers un objet de la classe Sortie.
  2. struct expOut

```
struct expOut
{
RWCString transName;
RWTPtrOrderedVector<Sortie>*    outL;        // Vecteur des sorties
RWTValOrderedVector<int>        vectSortie;  // Vecteur d'entiers
RWTValOrderedVector<int>        vectParam;   // Vecteur d'entiers
};
```

  RWTPtrOrderedVector<expOut> est un vecteur dont chaque élément est un pointeur vers une struture expOut.

- RWTValHashDictionary<T, V> : Classe RogueWave permettant de maintenir une table de hashage dont la clé est de type T et la valeur de type V.

Exemples :

1. RWTValHashDictionary<RWCString, int> est une table de hashage dont la clé est de type chaîne de caractères et la valeur de type entier.

Cette table de hashage a été utilisée, entre autre, pour contenir les variables d'état du modèle de la spécification, durant l'exécution du module Testeur qui à partir des données d'entrée, fournit les sorties attendues. Pendant cette exécution, les variables d'état sont mises à jour après chaque exécution d'une transition comportant une instruction d'action. La table de Hashage permet alors de sauvegarder et de récupérer les variables et leur valeurs d'une manière fiable, efficace et rapide.

3. RWTValHashDictionary<RWCString, RWTPtrOrderedVector<expOut>> est une table de hashage dont la clé est de type chaine de caractères et la valeur de type vecteur de struture expOut.

La structure expOut, décrite ci-dessus, contient une liste de transitions. A chaque transition est associé un vecteur de sorties attendues, pour chaque sortie la liste de ses paramètres, et un vecteur d'entiers qui indique pour chaque sortie si cette sortie et ses paramètres sont les mêmes que ceux qui ont été observés dans la trace d'exécution.

La table de hashage contient une liste de cas de tests. Pour chaque cas de test (le numéro de test est la clé) elle retourne la structure expOut qui exprime le résultat de la comparaison entre ce qui a été observé dans la trace d'exécution, et ce qui est attendu suit a une exécution faite sur le modèle de la spécification. À partir de cette table, on extrait les symptômes par cas de test.

9.5 Implantation du modèle de données de l'application

L'implantation du modèle de données s'est faite évidemment à partir de modèle objet d'application en respectant certaines règles de passage. Ainsi, chaque classe du modèle objet devient une classe implantée et chacune des relations s'est transformée ou bien en une classe ou bien en un pointeur.

Le détail du modèle implanté est donné dans l'ANNEXE B.

9.6 Implantation des modules de l'application

Tous les modules et leurs principales fonctions ont été décrits plus haut au paragraphe 9.2 Nous donnons un exemple d'implantation d'un des modules, le Testeur. Ce module prend les entrées de la trace d'exécution et les injecte dans le modèle de la spécification afin d'obtenir les sorties correctes attendues.

Le modèle de la spécification est un AEFE, étendu par les variables. Injecter une entrée revient alors à :

- rechercher la transition qui accepte l'entrée,
- évaluer la condition associée à la transition, compte tenu de l'état courant des valeurs des variables d'état,
- exécuter la transition, ses actions et mettre à jour la progression des états de AEFE,
- mettre à jour les valeurs des variables d'état,
- fournir les sorties ainsi que leurs paramètres munis de leur valeurs.

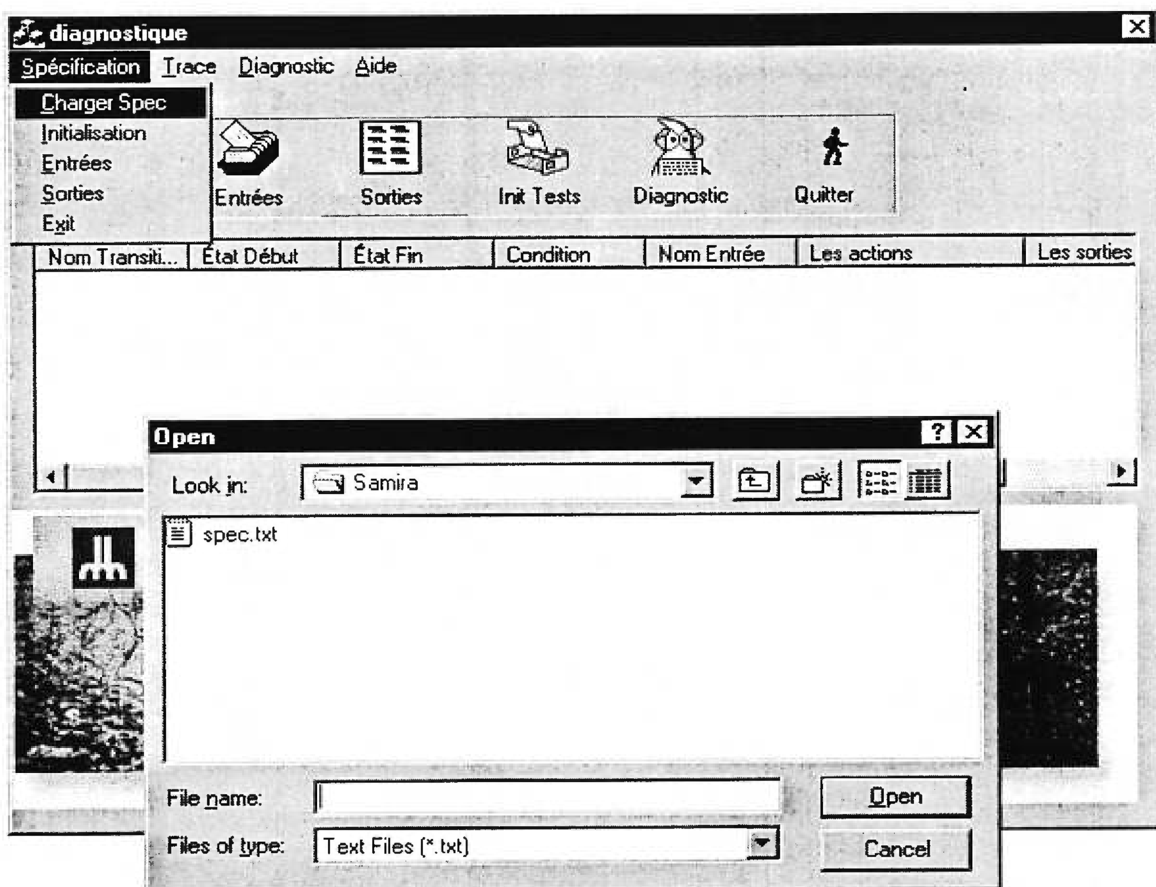
9.7 Interface du système

L'interface du système est constituée principalement des contrôles de la librairie MFC ainsi que des classes améliorées du MFC rendues publiques sur le site : www.codeguru.com

Chargement de spécification

Les spécifications sont fournies dans des fichiers textes selon un format spécifique. Le chargeur de spécifications transforme la spécification donnée au format interne. Pour charger une spécification :

- Cliquer sur Charger Spec dans le menu Spécification.
- Sélectionner le fichier de la spécification désirée.



Initialisation

Une fois le fichier des spécifications chargé et décortiqué par le système, la spécification est affichée dans le menu. Indiquant les noms des transitions, leurs états de début et de fin, leurs conditions de gardes, les actions, les entrées et les sorties. Pour initialiser une spécification :

- Cliquer sur Spec dans le menu spécification.

Le dialogue suivant s'affiche.

diagnostique

Spécification Trace Diagnostic Aide

Charger Spec
Initialisation
Entrées
Sorties
Exit

Entrées Sorties Init Tests Diagnostic Quitter

Nom Transiti...	État Début	État Fin	Condition	Nom Entrée	Les actions	Les sort
t1	Disconnected	Wait	-	ICONreq()	{Counter := 0}	cr()
t2	Wait	Connected	-	cc()	{Number := 1,Counte...	ICONcc
t3	Wait	Wait	Counter < 4	-	{Counter := Counter ...	cr()
t4	Wait	Disconnected	Counter = 4	-	{}	IDISind
t5	Connected	Send	-	IDATreq(IS...	{olddata := ISDU}	DT(Nur
t6	Send	Connected	Num = Num...	AK(Num)	{Number := Number ...	-
t7	Send	Send	Num <> Nu...	AK(Num)	{Counter := Counter ...	DT(Nur
t8	Send	Disconnected	Num <> Nu	AK(Num)	{}	IDISind

Université de Montréal

Les Entrées

Pour lister toutes les entrées de la spécification:

- Cliquer sur Entrées dans le menu de la spécification.

Un dialogue affichant l'entrée, ses paramètres et les transitions ou elle figure apparaît.

diagnostique

Spécification Trace Diagnostic Aide

Charger Spec
Initialisation
Entrées
Sorties
Exit

Entrées Sorties Init Tests Diagnostic Quitter

Nom Transiti...	État Début	État Fin	Condition	Nom Entrée	Les actions	Les sor
t1	Disconnected	Wait	-	ICONreq()	{Counter := 0}	cr()
t2	Wait	Connected	-	cc()	{Number := 1,Counte...	ICONcc
t3	Wait	Wait	Counter < 4	-	{Counter := Counter ...	cr()
t4	Wait	Disconnected	Counter = 4	-	{}	IDISind
t5	Connected	Send	-	IDATreq(IS...	{olddata := ISDU}	DT(Nur
t6	Send	Connected	Num = Num...	AK(Num)	{Number := Number ...	-
t7	Send	Send	Num <> Nu...	AK(Num)	{Counter := Counter ...	DT(Nur
t8	Send	Disconnected	Num <> Nu	AK(Num)	{}	IDISind

Université de Montréal

Liste des entrées / sorties

Nom Entree	Paramètres	Transitions
ICONreq		t1
cc		t2
IDATreq	ISDU	t5
AK	Num	t6, t7, t8
dr		t11, t12, t13, t14

Exit

Les Sorties

Pour lister toutes les sorties de la spécification:

- Cliquer sur Sorties dans le menu de la spécification.

Le dialogue suivant s'affiche.

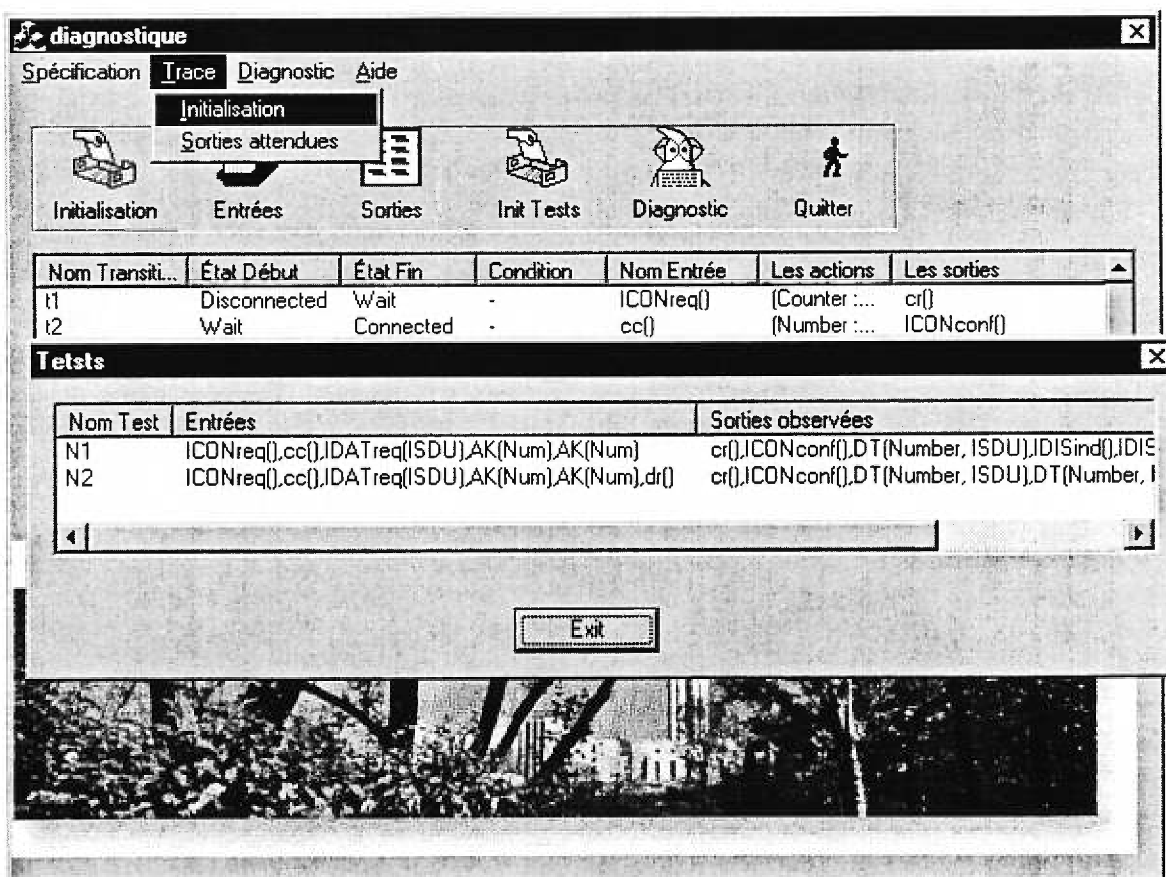
Initialisation de la trace d'exécution

Il s'agit de récupérer les traces d'exécution réalisées sur le système à tester. Cette trace est un ensemble de cas de tests numérotes (N1, N2,...). Chaque cas de test est une suite d'entrées/sorties. Désignant l'entrée fournie et la sortie recueillie après l'exécution de l'entree sur le système sous test.

Pour initialiser la base des tests :

- Cliquer sur initialisation dans le menu Trace.

Le dialogue suivant s'affiche.



Les symptômes

Les symptômes représente la différence entre les sorties attendues par le système et celle recueillies après l'application des tests sur le système sous test. Pour afficher les symptômes :

- Cliquer sur Symptôme dans le menu Diagnostic.

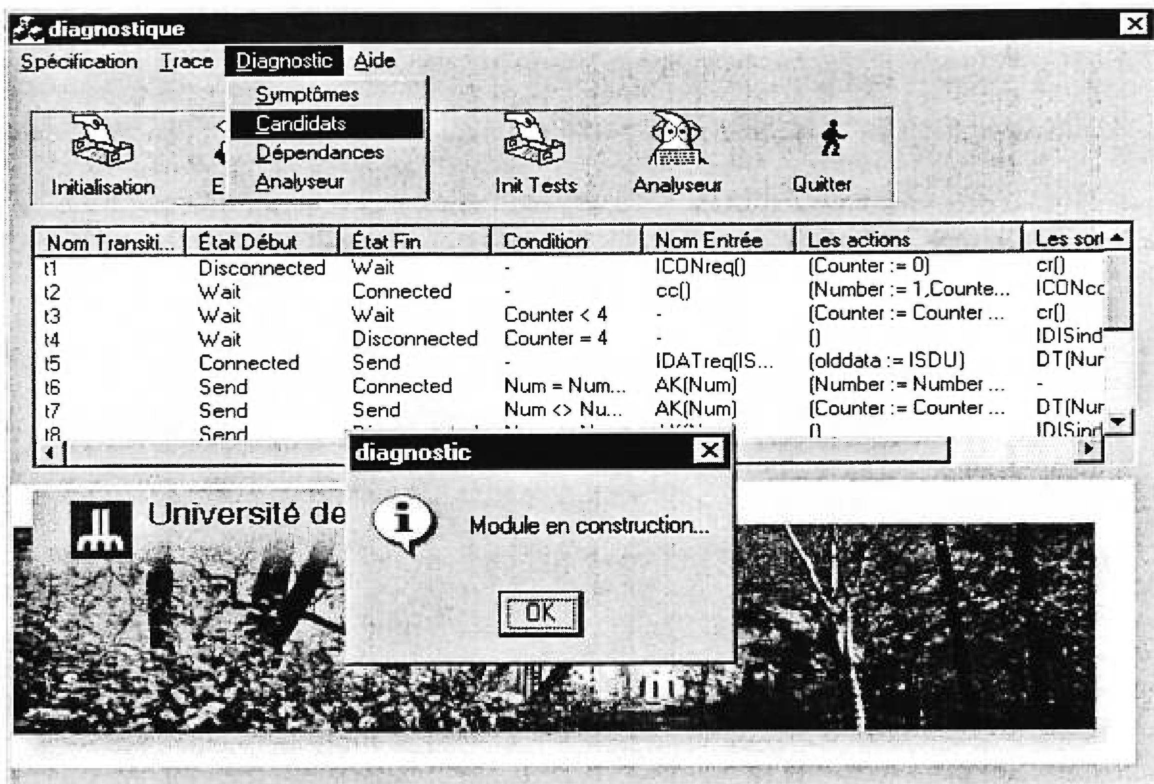
Le dialogue suivant s'affiche.

Les Candidats

Ce module n'est pas réalisé dans la version actuelle de l'outil .

- Cliquer sur Candidats dans le menu Diagnostic.

Le dialogue suivant s'affiche.

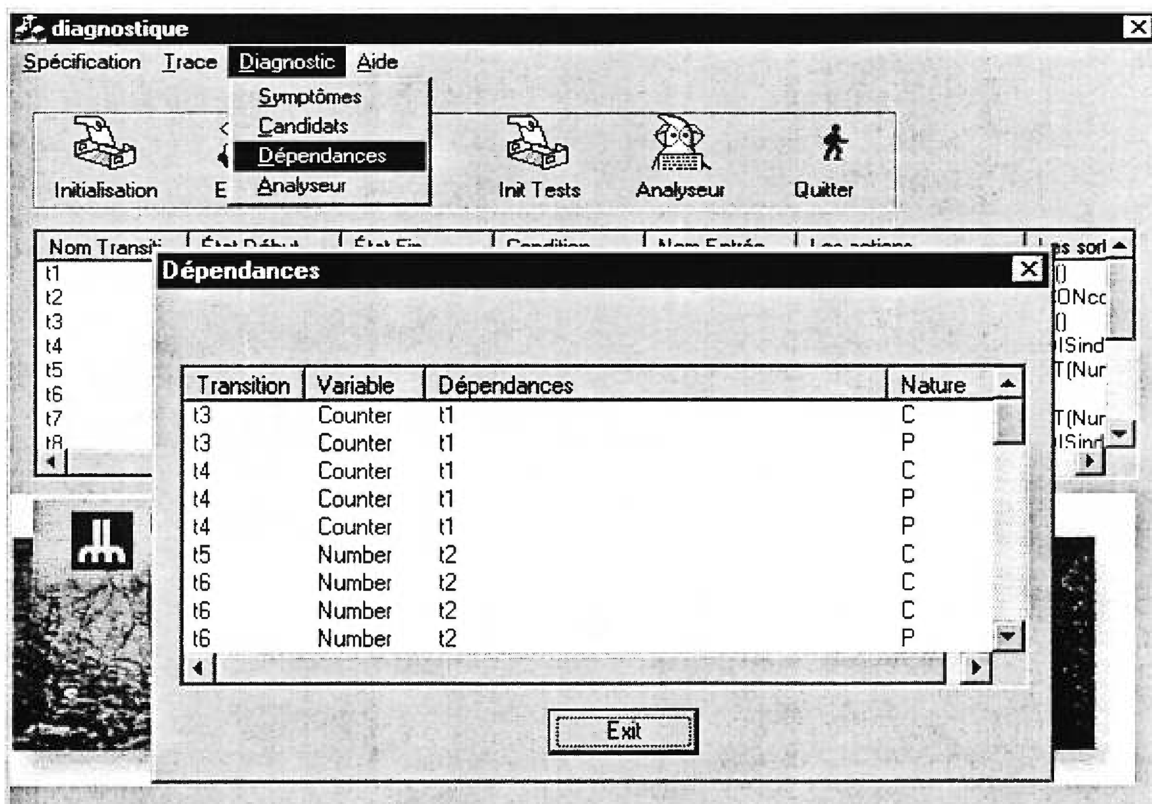


Les dépendances des transitions

Ce module détermine les dépendances entre les transitions du modèle de la spécification. Une fois le choix Dépendances sélectionné, le module Dépendance-Transition affiche toutes les dépendances Définition-Usage par rapport a toutes les variables de la transition et indique sa nature (C pour C-use ou données , et P pour P-use ou de contrôle).

- Cliquer sur Dépendances dans le menu Diagnostic.

Le dialogue suivant s'affiche.

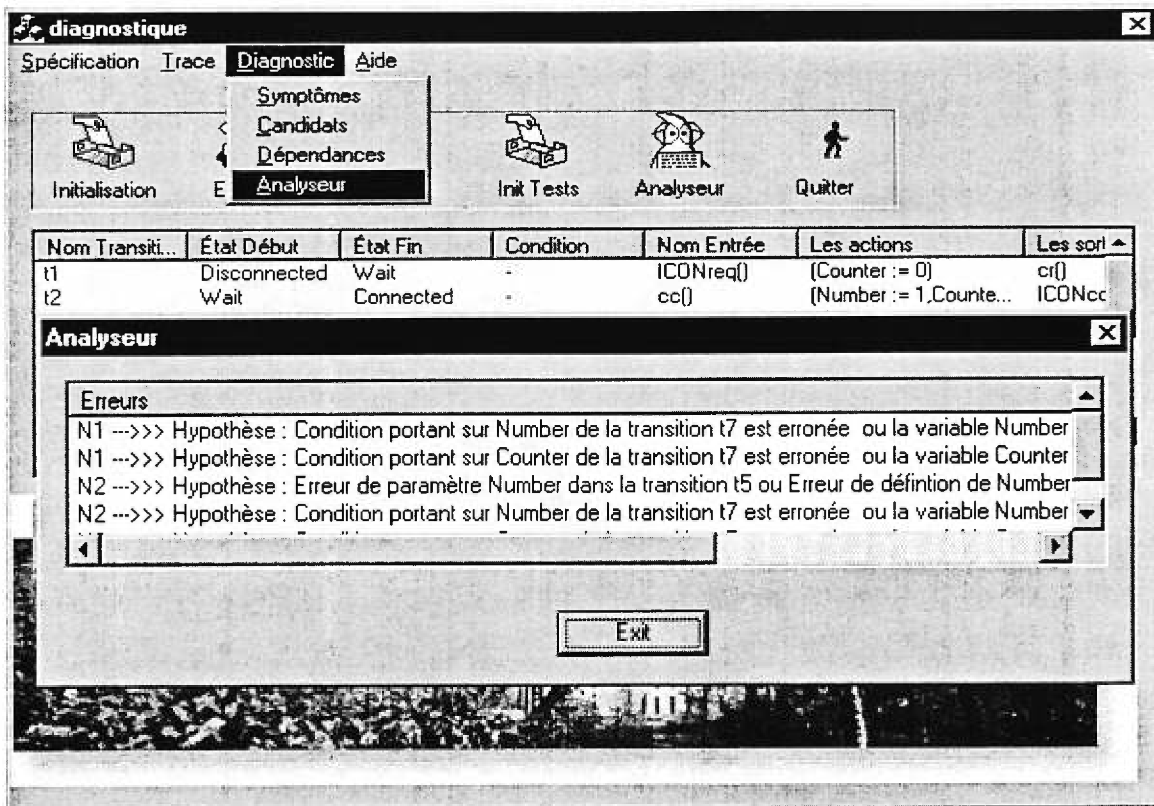


L'Analyseur

A partir des symptômes et des dépendances des transitions, l'analyseur affiche les hypothèses sur les transitions fautives. Les hypothèses sont formulées par cas de test, elles indiquent la transition fautive et la cause de la faute suspectée.

- Cliquer sur Analyseur dans le menu Diagnostic.

Le dialogue suivant s'affiche.

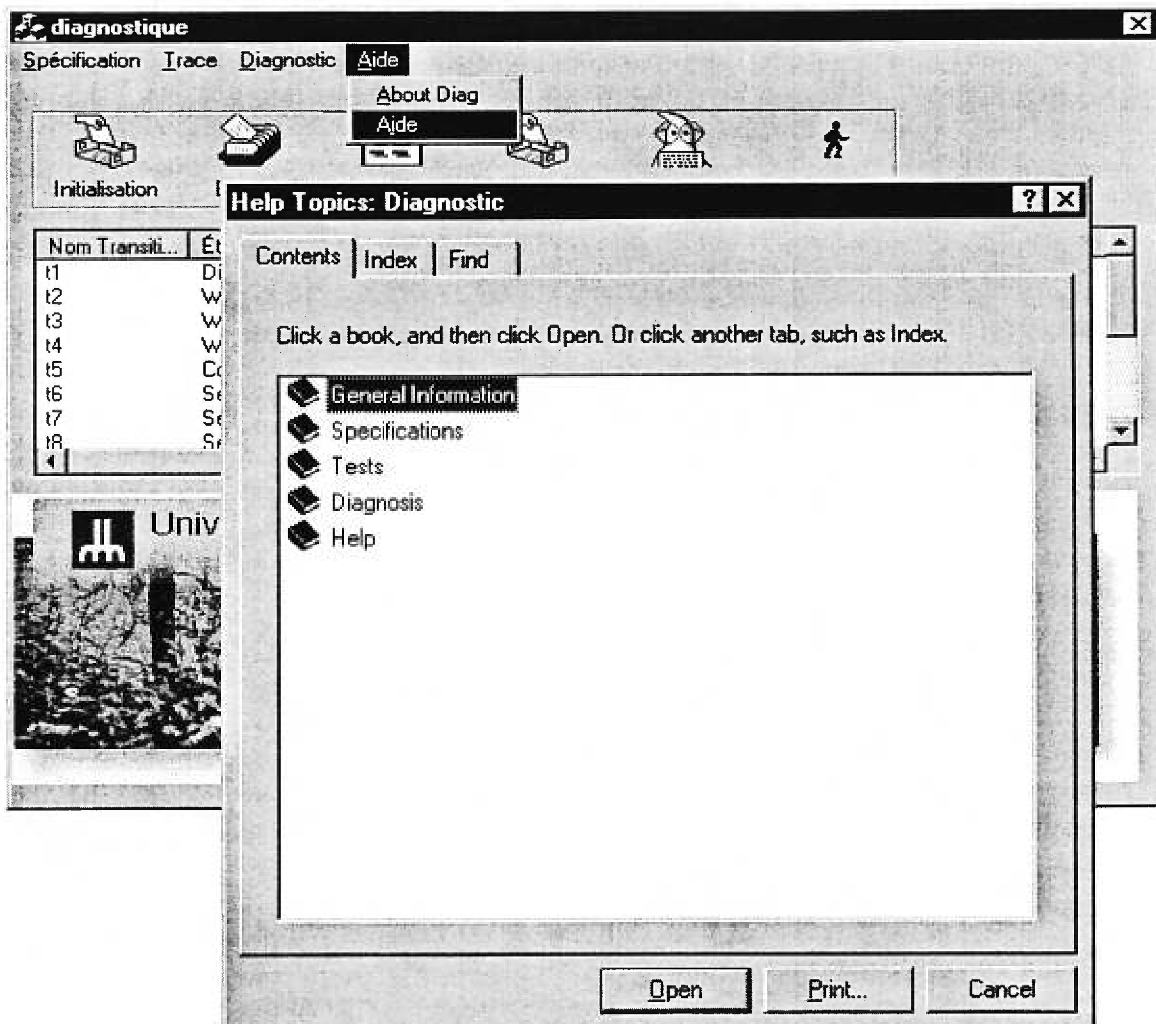


L'Aide

Pour afficher l'aide :

- Cliquer sur Aide dans le menu Aide.

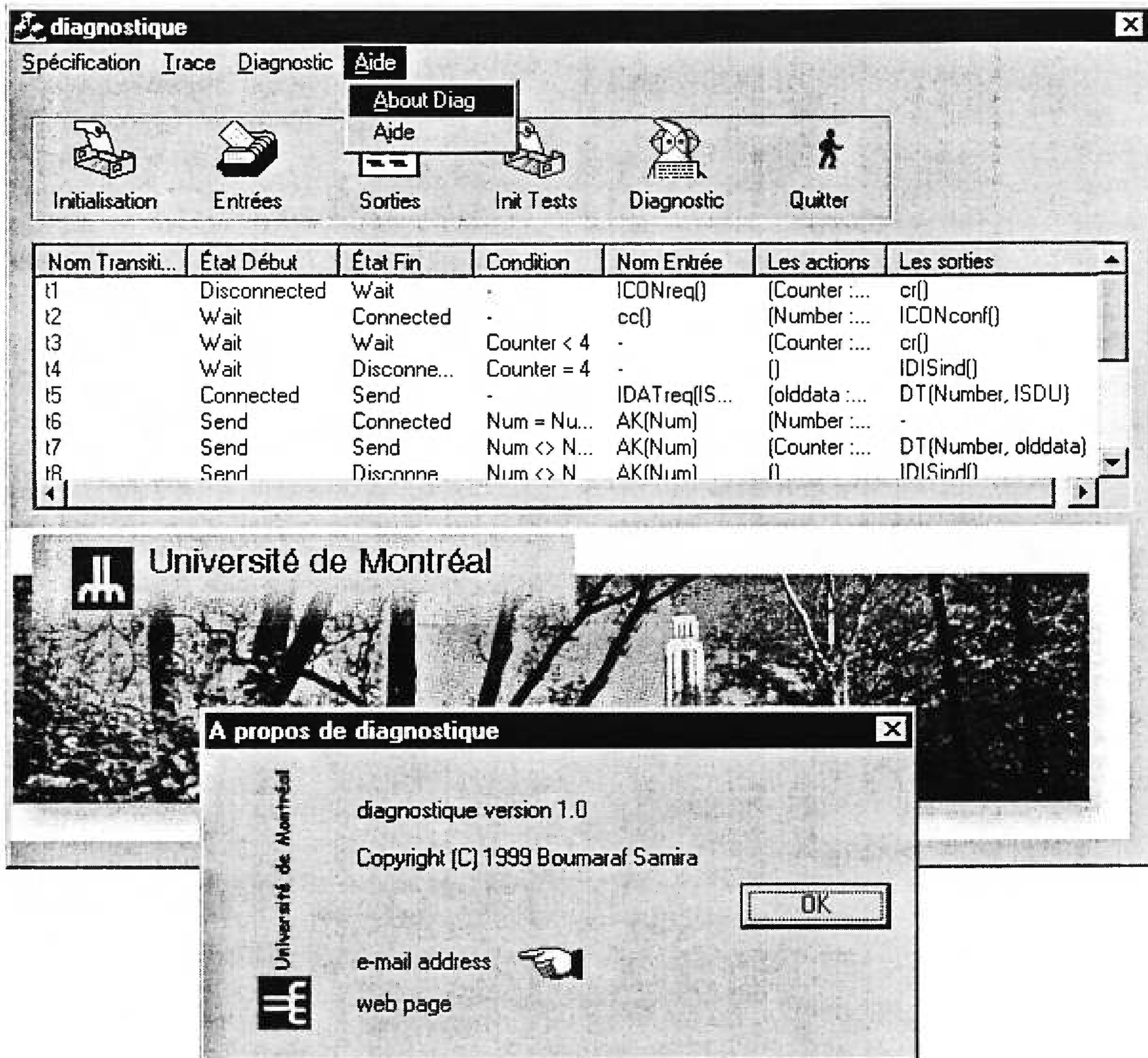
Le dialogue suivant s'affiche.



About diagnostic

Cliquer sur About Diag dans le menu de Aide.

Le dialogue suivant s'affiche.



Conclusion

Le diagnostic basé sur modèle (model-based diagnosis) se classe parmi les plus importantes approches pour traiter le problème du diagnostic des protocoles de communication, grâce à sa généralité, sa flexibilité et sa fiabilité. Selon le formalisme utilisé pour décrire la spécification, une variété de techniques peuvent être utilisées pour dériver des suites de tests ou pour effectuer le processus de diagnostic. Dans de nombreux cas, une distinction est faite entre l'aspect contrôle et l'aspect données du système à tester. Le flux de contrôle est souvent décrit par un Automate à États Finis (AEF), qui sert de modèle de référence pour le test et le diagnostic. Cependant pour des systèmes réels, le test du contrôle doit être complété par le test du flux de données, souvent traduit par un Automate à États Finis Étendu (AEFE).

Dans ce mémoire, nous avons passé en revue certaines techniques de diagnostic fondées sur un modèle et en particulier celles appliquées dans le domaine de protocoles de communication. Nous avons proposé une approche de diagnostic fondée sur l'usage des automates à états finis étendus par les variables. Notre contribution concerne le développement d'une méthode de diagnostic pour le flux de données de l'automate. Cette méthode utilise le graphe de dépendances des transitions qui est parcouru des feuilles vers la racine ce qui est en effet une adaptation de l'algorithme de Chanson [Chan 93]. Le critère définition-usage a été utilisé, il permet pour un usage de transition donné de remonter aux définitions des variables de l'usage.

Nous avons également développé un outil EMFDT qui implante la méthode de diagnostic proposée. EMFDT est un outil automatique de diagnostic. Il permet, à partir d'une spécification de protocole (écrite dans un langage spécifique) et d'une trace d'exécution de suite des tests, de générer des diagnostics à propos des transitions fautives d'une implémentation sous test.

Ce travail nous a montré qu'il n'était pas évident de retrouver une cause précise de l'erreur quand on manipule des valeurs de variables. En effet on arrive toujours à détecter quelle est la variable qui induit l'erreur, mais retrouver la valeur de la variable qui induit

l'erreur n'est pas une tâche facile, en particulier si les variables sont définies dans un grand intervalle de valeurs.

Nous suggérons de compléter l'outil EMFDT par un module de génération de candidats. Ce module se servira des hypothèses générées par la version actuelle de l'outil pour générer des candidats diagnostics. L'intégration de nouvelles stratégies de sélection de valeurs des variables enrichirait l'outil et pourrait peut-être pallier aux problèmes de sélection rencontrés.

Références

[Aho 88] A. V. Aho, A. T. Dahbura, D. Lee, M. U. Uyar, "An Optimization technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours", Protocol Specification, Testing and Verification VIII, IFIP, 1988.

[Barr 91] S. B. Barry, U. Uyar, " Finite state machine based formal methodes in protocol conformance testing: from theory to implementation. Computer Networks and ISDN Systems 22, pp. 7-33, 1991.

[Besc 93] A. Beschta, O. Dressler, H. Freitag, M. Montag, P. Strauss, "A model-based approach to fault localisation in power transmission networks", Intelligent Systems Engineering, Spring, 1993.

[Boch 77] G. v. Bochmann, J. Gecsei, "A unified method for the specification and verification pf protocols", in B. Gilchrist, editor, information processing 77, pp. 229-234, IFIP, 1977.

[Boch 78] G.v. Bochmann, "Finite state description of communication protocols", Computer Networks, Vol. 2, No. 4/5, 1978.

[Boch 83] G.v. Bochmann, "Testing transport protocol implementations", Proc. of CIPS Conference 1983, Ottawa, 1983, pp. 123-129.

[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 (invited paper), pp. (II-17)-(II-32).

[Boul 90] A. T. Bouloutas, "Modelling Fault Management in Communication Networks", PhD Thesis, Columbia University, 1990.

[Bour 97] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, "Automatic Executable Test case Generation for Extended Finite State Machine Protocols", International Workshop Testing of Communicating Systems, IWTCS'97, September 97.

[Bour 99] C. Bourhfir, "Génération automatique de cas de test pour les systèmes modélisés par des machines à états finis communicantes", thèse de PhD, Université de Montréal, Avril 1999.

[Chan 93] S.T Chanson, J. Zhu, "A Unified Approach to Protocol Test Sequence Generation", Proc. IEEE INFOCOM.

[Davi 82] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, S. Polit, "Diagnosis Based on Description of Structure and Fonction", Proceedings AAAI, pp. 137-142, 1982.

[Davi 84] R. Davis, "Diagnostic reasoning based on structure and behavior", Artificial Intelligence 24 (3), pp. 347-410, 1984.

[Davi 88] R. Davis, W. Hamscher, "Model-Based reasoning : Troublesshooting", Artificial Intelligence, Shrobe, H.E, The American Association for Artificial Intelligence (eds), pp. 297-346, Morgan Kaufman, 1988.

[DeMe 91] R. A. DeMillo, J. Offutt, " Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, vol. 17, no 9, 1991.

[Derr 95] K.W. Derr, "Applying OMT", journal of object oriented programming Report on Object Analysis and Design, SIGS Publications, Inc. New York.

[Dsso 99] R. Dssouli, K. Saleh, El M. Aboulhamid, A. En-Nouaary and C. Bourhfir,"Test development for communication protocols: Toward automation", in Computer Networks, The International Journal of Computer and Telecommunications Networks, Special Issue ISSN 1389-1286 , Advanced Topics on SDL and MSC, Vol. 31, Num. 17, June 1999, Elsevier, pp.1835-1872.

[Fuji 91] S. Fujiwara, G. V. Bochmann, F. Khendek, A. Amalou, A. Ghedamsi, "Test selection based on finite state models", IEEE Transaction on Software Engineering, vol.17 no 6, pp. 591-603, june 1991.

[Ghed 93a] A. Ghedamsi, "Diagnostic Tests for Protocol Implementations Modeled by Finite State Machines", PhD Thesis, 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.

[Gene 84] M.r. Genesereth, "The use of design descriptions in automated diagnosis", Artificial Intelligence 24 (3), pp. 411-436, 1984.

[Gon 70] G. Gonenc, " A method for the design of fault detection experiments", IEEE Trans. Computer, vol. C-19, pp. 551- 558, june 1970.

[Khen 90] F. Khendek, "Diagnostic basé sur un modèle", Rapport technique, Dept. IRO, Université de Montréal, Avril, 1990.

[Kime 86] C. R. Kime, "System Diagnosis", volume 2, chapter 8, pp. 577-632, Prentice Hall, 1986.

[Klee 87] J. de Kleer, B.C Williams, "Diagnosing multiples faults", Artificial Intelligence 32(1), 1987, pp. 97-130.

[Klee 89] J. de Kleer, B.C Williams, "Diagnosing with Behavioral Modes", Proceedings IJCAI, Detroit-Michigan, 1989, pp. 1324-1330.

[Knig 88] K.g Knightson, T. Knowles, J. Lrmouth, " Standards for Open systems interconnections", McGraw-hill, 1988.

[Koha 78] Z. Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, second edition,1978.

[Kuhl 80] J. G. Kuhl, "Fault Diagnosis in Computing Networks", PhD thesis, university of Iowa, 1980.

[Hams 92] W. Hamscher, L. Console, J. de Kleer, editors, "Readings in Model-Based Diagnosis", Morgan-Kaufmann, 1992.

[Hart 87] G. W. Hart, "Minimum Information Estimation of Structure", PhD thesis, MIT, 1987.

[Holz 91] G. Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall, 1991.

[Holz 93] G. Holzmann, "Design and validation of protocols: a tutorial", Computer Networks and ISDN Systems, 25, pp. 981-1017, 1993.

[Htit 97] S. Htite, "Diagnostic automatique avec l'outil MFDT ", CIFIP'97, 1997, pp. 287-300.

[ISO 9646] ISO-Information Processing Systems-Open Systems Interconnection- OSI Conformance Yesting Methodology and Framework, Parts 1, 2, 3, 4, 5, ISO 9646.

[Lewi 93] L. Lewis, "A case-based reasoning approach to the resolution of faults in communication networks, in Integrated Network Managment, III, pp. 671-682, Elsevier Publishers B. V. (North-Holland), 1993.

[Lin 91] Y. J. Lin, G. Wuu, "A constraint approach for temporal intervals in the analysis of timed transitions", Protocol Specification, Testing and Verification, XI, PP. 215-230, October 1991.

[Math 87] R. Mathonet, H. V. Cottem, L. Vanryckeghem, "DANTES: an expert system for real-time network troubleshooting", in IJCAI, pp. 527-530, 1987.

[Merl 79] P.m. Merlin, "Specification and validation of protocols", IEEE Trans. on Communications, Vol. COM-27, No.11, 1979, pp. 1671-1679.

[Mill 92] R. E. Miller, S. Paul, "Generating Conformance Test Sequence for Combined Control and Data Flow of Communication Protocols", Proceeding of Protocol Specification, Testing and Verification (PSTV' 92), Florida, USA, june 1992.

[Miln 80] R. Milner, "A calculus of communicating systems", Lecture Notes in Computer Science 92, 1980.

[Nait 81] S. Naito, M. Tsunoyama, "Fault Detection for Sequential Machines by Transition-Tours", Proceedings of FTCS (Fault Tolerant Computing Systems), pp. 238-243, 1981.

[Prep 67] F. P. Preparata, G. Metze, R. T. Chien, "on the connection assignment problem of diagnosible systems", IEEE Transactions on Electronic Computers, EC-16, pp.848-854, December 1967.

[Rayn 87] D. Rayner, "Standardizing Conformance Testing for OSI", Computer Networks and ISDN Systems, Vol.14, No. 1, 1987.

[Ries 93a] M. Riese, "Modele-Based Diagnosis of Communication Protocols", PhD Thesis, EPFL, Suisse, 1993.

[Ries 93b] M. Riese, "Réalisation d'un système de diagnostic basé sur modèle pour les protocoles de communication", CIFIP'93, 1993.

[Reit 87] R. Reiter, "A theory of diagnosis from first principles", Artificial Intelligence 32 (1), pp. 57-96, 1987.

[Rumb 91] J. Rumbaugh, "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[Sabn 85] K.K. Sabnani, A. T. Dahbura, "A new Technique for Generating Protocol Tests", ACM Computer Communication Revue, vol. 15, no 4, 1985.

[Sabn 88] K.k Sabnani, A. T. Dahbura, " A protocol Testing Peocedure", Computer Networks and ISDN systems, vol. 15 no 4, pp. 285-297, 1988.

[Sari 87] B. Sarikaya, G.v. Bochmann,, E. Cerny, " A Test design Methodology for Protocol Testing", IEEE Trans. on SE, April 87, pp. 518-531.

[Sari 89] B. Sarikaya, "Conformance Testing: Architecture and Test Sequences", Computer Networks and ISDN Systems 17, pp. 111-126, 1989.

[Schw 82] R.l. 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.

[Shap 92] S. Shapiro, editor, Encyclopedia of Artificial Intelligence, Wiley, second edition, 1992.

[Shaw 92] A. Shaw, "Communicating real-time state machines", IEEE Transactions on Software Engineering, 18 (9), pp. 805-816, September 1992.

[Shir 83] M. Shirley, R. Davis, "Generating Distinguishing Tests Based on Hierarchical Models and Symptom information", Proceedings of the IEEE International conference on Computer Design, 1983.

[Stru 89] P. Struss, O. Dressler, "Physical Negation- Integrating Fault Models into the General Diagnostic Engine", in : Proceedings IJCAI, Detroit-Michigan, pp. 1318-1323, 1989.

[Stru 90] P. Struss, "Qualitative Models Diagnostic Systems", in : Carnevale, M.,Lucertini. M., and Nicosia, S. (eds), Proceedings of the IFIP TC7 conference on Modelling the innovation, Rome, 1990.

[Ural 87] H. Ural, "A Test Derivation Method for protocol Conformance Testing", Proc. of the 7th IFIP Symposium on Protocol Specification, Testing and Verification, Zurich, May 5-, 1987.

[Ural 91] H. Ural, B. Yang, "A Test Sequence Selection Method for Protocol Testing", IEEE Transactions on Communication, vol. 39, no 4, April 91.

[Vuon 89] S.t. Vuong, W. Chan, M.r. Ito, " The UIOv-Method for protocol test sequence generation", in the 2-nd International Workshop on Peotocol Test Systems, Berlin, Germany, October 3-6, 1989.

[Vuon 90] S.t. Vuong, K.C. Ko, "A novel approch to protocol test sequence generation", IEEE Global telecomm. conference and exhibition, San Diago, California, Dec. 2-5, 1990, vol no 3, 904.1.1-904.1.5.

[Wang 92] C. J. Wang, M. T. Liu, "A test suite generation methode for extended finite state machine using axiomatic semantics approach. Protocol Specification, Testing and Verification, XII, pp. 29-43, 1992.

ANNEXE A

Extrait de la forme de la spécification en entrée

```
FSMNAME      Protocole1

INITSTATE    disconnected

STATES       disconnected wait connected send ;

TRANSITIONS

When iconreq
From disconnected
To wait
Name t1:Begin
counter:=0;
Output(cr);
End;

When cc
From wait
To connected
Name t2: Begin
number:=1;
counter:=0;
Output(iconconf);
End;

From wait
To wait
Provided (counter<4)
Name t3: Begin
Output(cr);
counter:=counter +1;
End;

ENDFSM
```

ANNEXE B

Détails des classes implantées

class Transition

```
{
 public:
	//default constructor
	Transition();
	~Transition();
	RWCString GetName(){return Name;};
	void SetName(RWCString Nm){Name = Nm;};
	Entree *GetInput(){return Input;};
	void SetInput(Entree *Ent){Input= Ent;};
	void ConstrSortie(Sortie *Output);
	void ConstrAct(Action *Act);
	RWTPtrOrderedVector<Sortie>* GetListSrt(){return ListSorties;}; // Sorties d'une transition
	void	SetListSrt(SortieList *S){ ListSorties = S;};
	Etat* GetEtatD(){return EtatDeb;};		// Fonction retournant l'état début d'une transition
	Etat* GetEtatF(){return EtatFin;};		// Fonction retournant l'état fin d'une transition
	void SetEtatD(Etat* etatact){ EtatDeb = etatact;};
	void SetEtatF(Etat* etatact){ EtatFin = etatact;};
	Condition *GetCond(){return Cond;};		// Fonction retournant la condition d'une transition
	void	SetCond(Condition *C){ Cond = C;};
private:
	RWTPtrOrderedVector<Action>*	ListActions;	// les actions d'une transition
	int				numTrans;
	RWTValHashDictionary<RWCString, depend>*	DependT; // Dépendances d'une transition
	// depend est une structure :
	// struct depend	{
	//		BOOL				type;
	//		RWTPtrOrderedVector<Transition>*	transList;}; la liste des transition
	dont dépend la transition numTrans.

	RWCString			Name;		// Nom de la transition
	RWTPtrOrderedVector<Sortie>*	ListSorties;	// Liste des sorties d'une transition
	Entree*				Input;		// Entrée d'une transition
	Condition*			Cond;		// Condition (simple ou composée)
	Etat*				EtatDeb;	// État début d'une transition
	Etat*				EtatFin;	// État fin d'une transition
};
```

class Sortie

```
{
 public:
	Sortie();		// Constructeur par défaut
	~Sortie();		// Destructeur
	Sortie(int num, RWCString val);	// Constructeur avec paramètres
	Sortie(RWCString val){Valeur = val; ListParametres = NULL;};
	void ConstrParam(Parametre *P);
	RWCString				Valeur;		// Nom sortie
	int					numSor;		// Valeur sortie
	RWTPtrOrderedVector<Parametre>*		ListParametres;	// Liste des paramètres
};
```

```
class Entree
{
 public:
        Entree();
        ~Entree();
        Entree(int num, RWCString val);
        Entree(RWCString val){Valeur=val;ListParametres = NULL;ListTrans = NULL;};
        void ConstrParam(Parametre * P);
        void ConstrTrans(Transition *Trans);
        RWTPtrOrderedVector<Parametre>*GetListParam(){return ListParametres;};
        void    SetListParam(ParamList *S){ListParametres=S;};
        RWTPtrOrderedVector<Transition> *GetListTrans(){return ListTrans;};
        void    SetListTrans(TransList *T){ListTrans=T;};
        RWCString    Valeur;
        int                     numEnt;
 private:
        RWTPtrOrderedVector<Parametre>*        ListParametres;  // Liste des paramètres d'une entrée
        RWTPtrOrderedVector<Transition> *      ListTrans;       // Liste des transitions d'une entrée
};


class Action
{
 public:

        Action();
        Action(RWCString oper);
        ~Action();

        RWCString Operateur;     // Opérateur d'une action  (:=, >, ..)
        Terme* TermeG;          //  Terme gauche d'une action
        Terme* TermeD;          //  Terme droit d'une action
};

class Terme
{
 public:
        Terme();
        ~Terme();
        int Type;           // le type désigne si le terme est une variable, constante ou expression
};

class Condition
{
 public:
        virtual ~Condition();
        Condition();
        int Ccomp;          // indique si la condition est composée ou simple
};

class CondSmp:public Condition           // La classe Condition_simple hérite de Condition
{
 public:
        ~CondSmp();
        CondSmp();
        CondSmp(RWCString Cmp);
```

```
        RWCString GetComp(){return Comparateur;};
        void SetComp(RWCString Comp){Comparateur = Comp;};
        Terme* GetTermeD(){ return TermeD;};
        Terme* GetTermeG(){ return TermeG;};
        void  SetTermeG(Terme *terme){ TermeG = terme;};
        void  SetTermeD(Terme *terme){ TermeD = terme;};
private:
        RWCString      Comparateur;    // Comparateur
        Terme          *TermeG;        // Terme gauche de la condition
        Terme          *TermeD;        // Terme droit de la condition
};
```

```
class CondComp:public Condition        // La classe Condition_composée hérite de Condition
{
public:
        ~CondComp(); CondComp();
        CondComp(RWCString Op){OpLog = Op;};
        RWCString      OpLog;          // Opérateur logique de la condition composée (&, ou)
        Condition      *ExpD;          // Expression droite de la condition composée
        Condition      *ExpG;          // Expression gauche de la condition composée
};
class Variable: public Terme      // La classe Variable hérite de Terme
{
public:
        Variable();
        ~Variable();
        Variable(RWCString var);
        RWCString varble;         // Nom de la variable
};
```

```
class Constante: public Terme     // La classe Constante hérite de Terme
{
public:
        Constante() ;
        ~Constante();
        Constante(int Cste);
        int  Conste ;             // Valeur de la constante
};
```

```
class Expression: public Terme   // La classe Expression hérite de Terme
{
public:
        Expression();
        ~Expression();
        Expression(RWCString Op);
        RWCString OpArithm ;    // Opérateur arithmétique de l'expression  ( &, ..)
        Terme *Membre1;         // Membre gauche de l'expression
        Terme *Membre2;         // Membre droit de l'expression
};
```

```
class Etat
{
public :
        Etat();
        ~Etat();
        Etat(RWCString Nm);
```

```
        RWTPtrOrderedVector<Etat>*     ConstrEtat(RWCString Nm, EtatList *ListEtat);
        RWCString GetName()            {return Name;};         // Fonction retournant l'état
        void SetName(RWCString name)   {Name = name;};         // Fonction initialisant l'état
 private :
        RWCString Name;          // Nom de l'état
 };
```

class Parametre

```
{
 public:
        Parametre();
        ~Parametre();
        Parametre(int num, RWCString p, int V);
        Parametre(RWCString p, int V){Val = V; Param = p;};
        int   numParam;
        RWCString Param;         // Nom du paramètre
        int   Val;               // Valeur du paramètre
 };
```

class Test

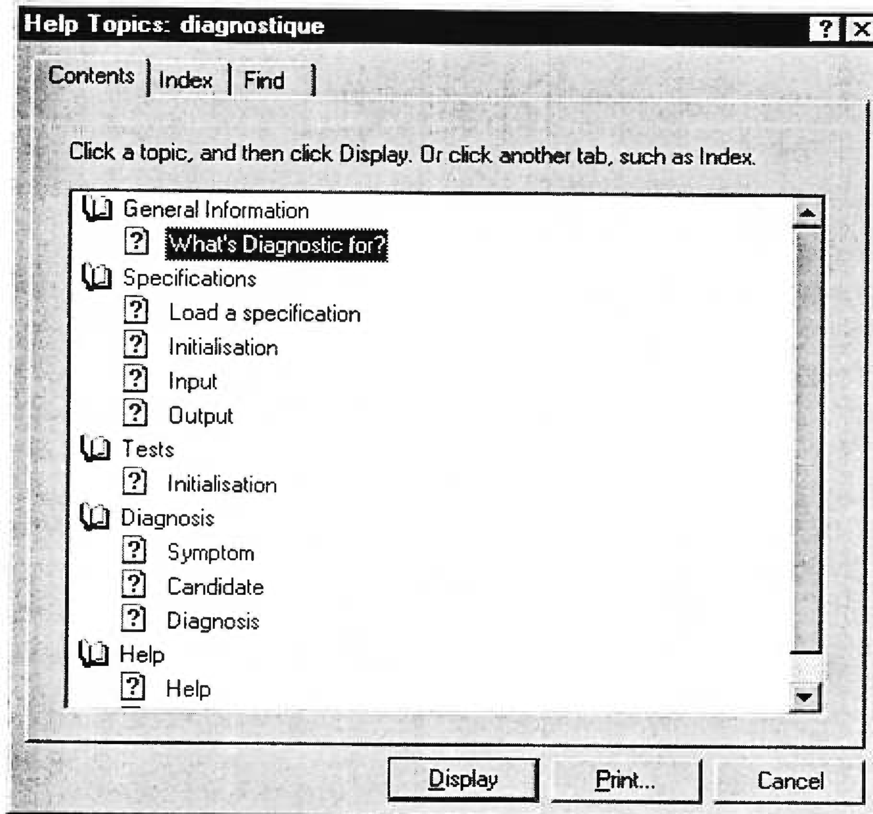
```
{
public:
        Test();
        ~Test();
        Test(RWCString tnum);
        RWCString GetTestNum(){return TestNum;};
        void SetTestNum(RWCString tnum){TestNum = tnum;};
        RWTPtrOrderedVector<Entree>*       GetListEnt(){return ListEntrees;};
        RWTPtrOrderedVector<Sortie>*       GetListSort(){return ListSorties;};
        void ConstrEntree(Entree *Ent);
        void ConstrSortie(Sortie * Out);
private:
        RWCString                      TestName;          // Nom  du test
        RWTPtrOrderedVector<Entree>*   ListEntrees;       // Liste des entrées du test
        RWTPtrOrderedVector<Sortie>*   ListSorties;       // Liste des sorties du test
};
```

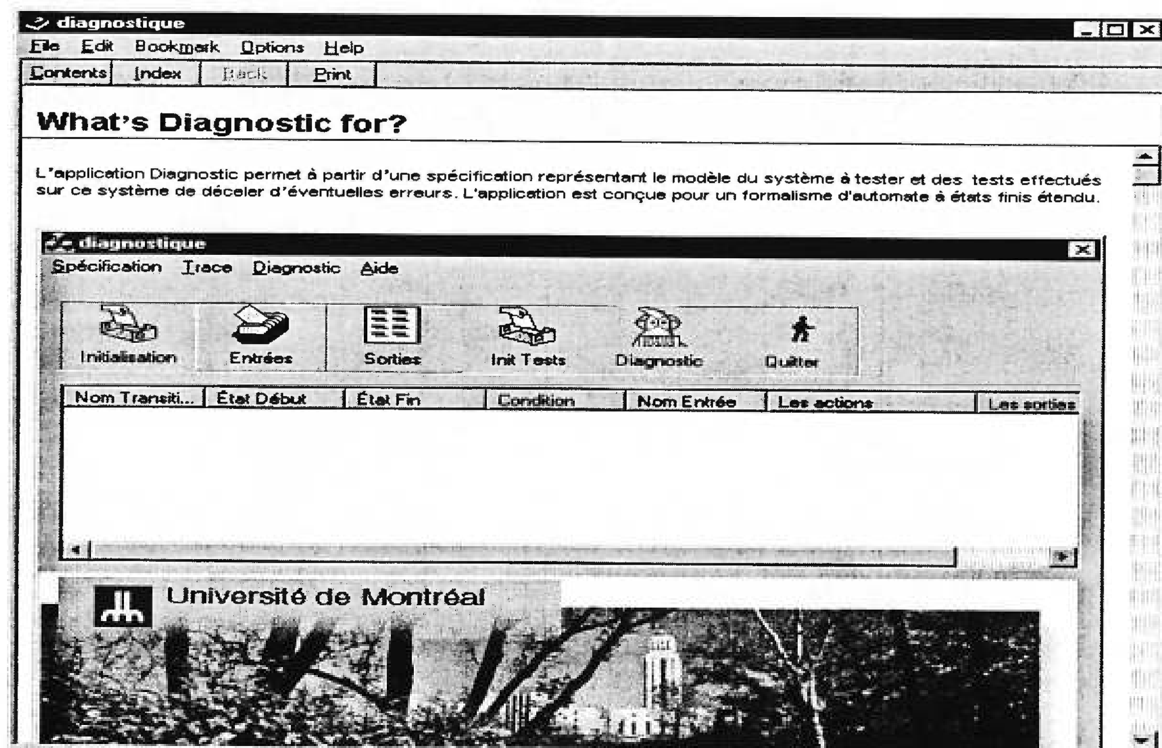
class Trace

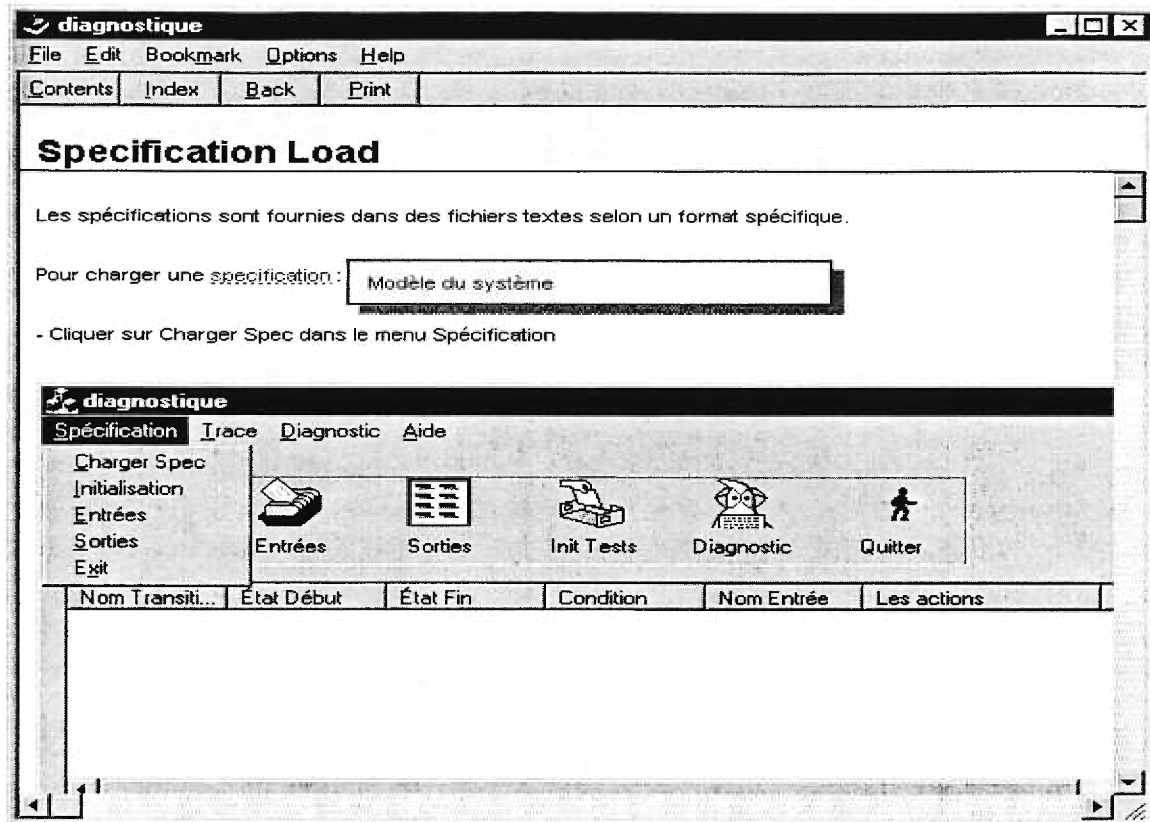
```
{
public:
        Trace();
        ~Trace();
        Trace(RWCString tsuite);
        RWTPtrOrderedVector<Test>*    ConstrTest(char * val, TestList *ListTests);
        bool operator ==(const Trace & anotherTrace);

        RWCString                     TestSuite;      // Nom de la trace
        RWTPtrOrderedVector<Test>*    ListTests;      // Liste des tests
};
```

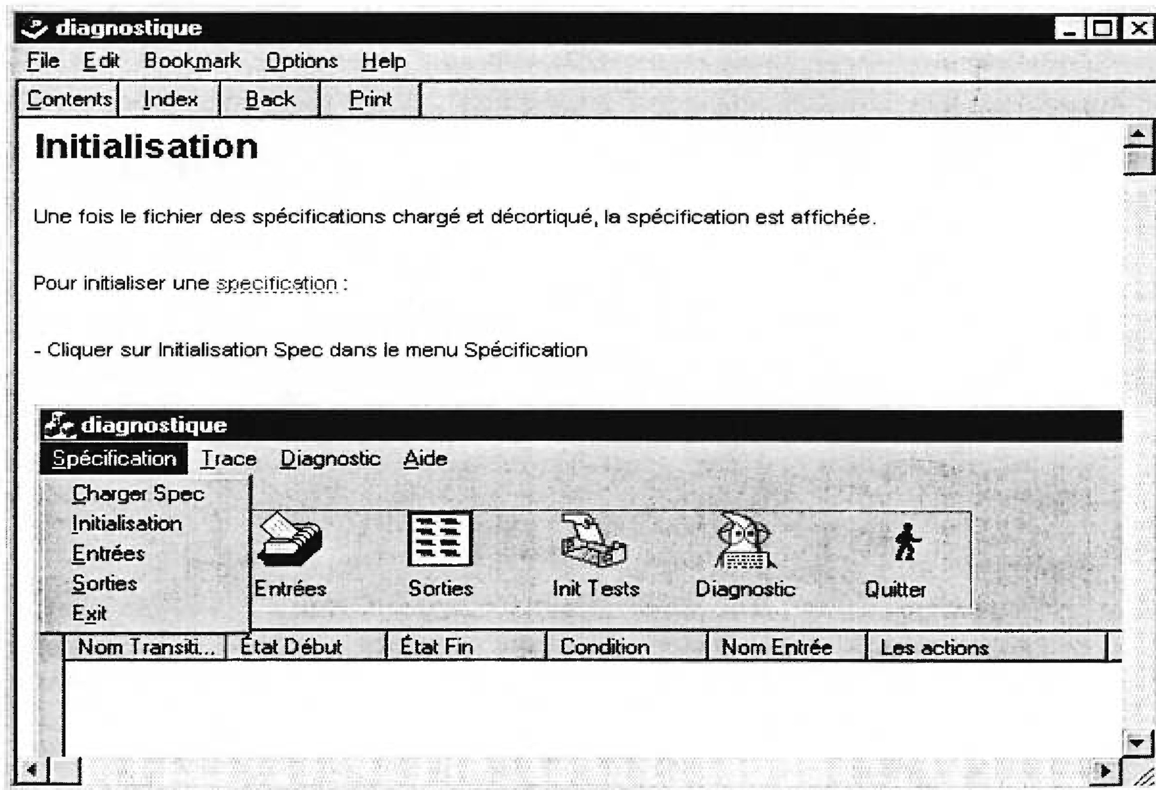
ANNEXE C

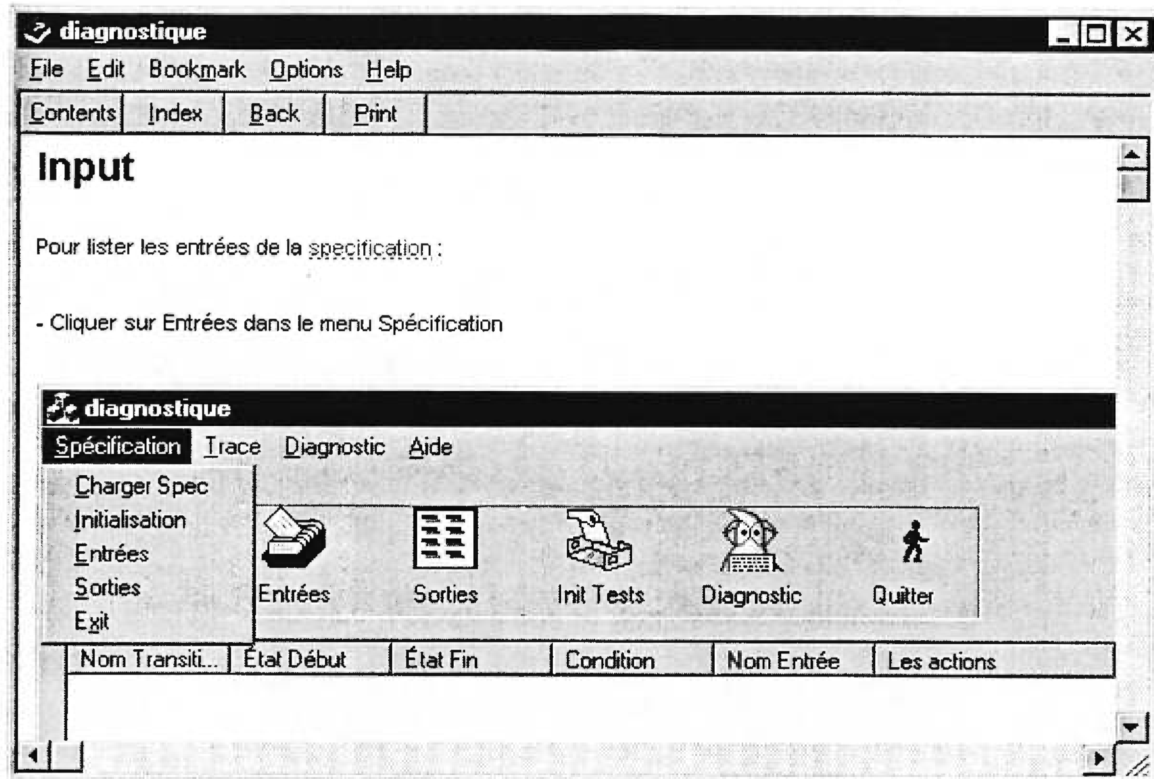
Cette annexe décrit l'aide fournie par le système à l'utilisateur de l'outil EMFDT.

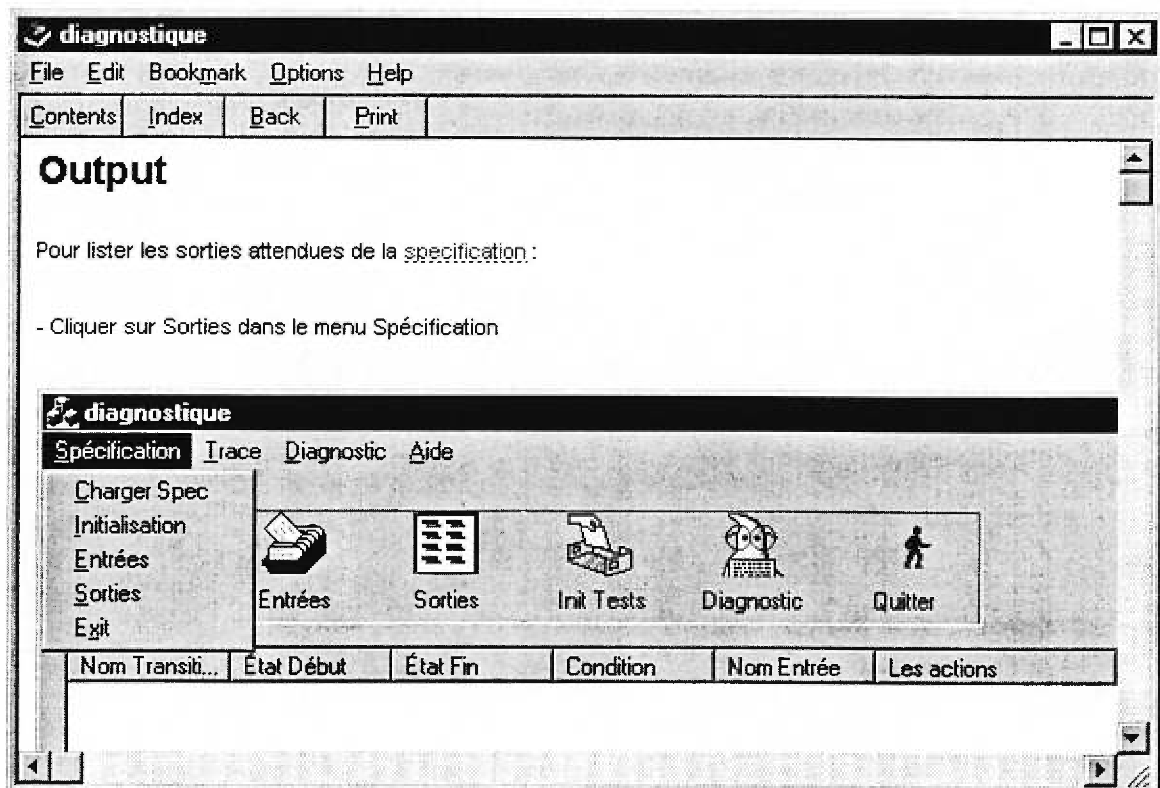


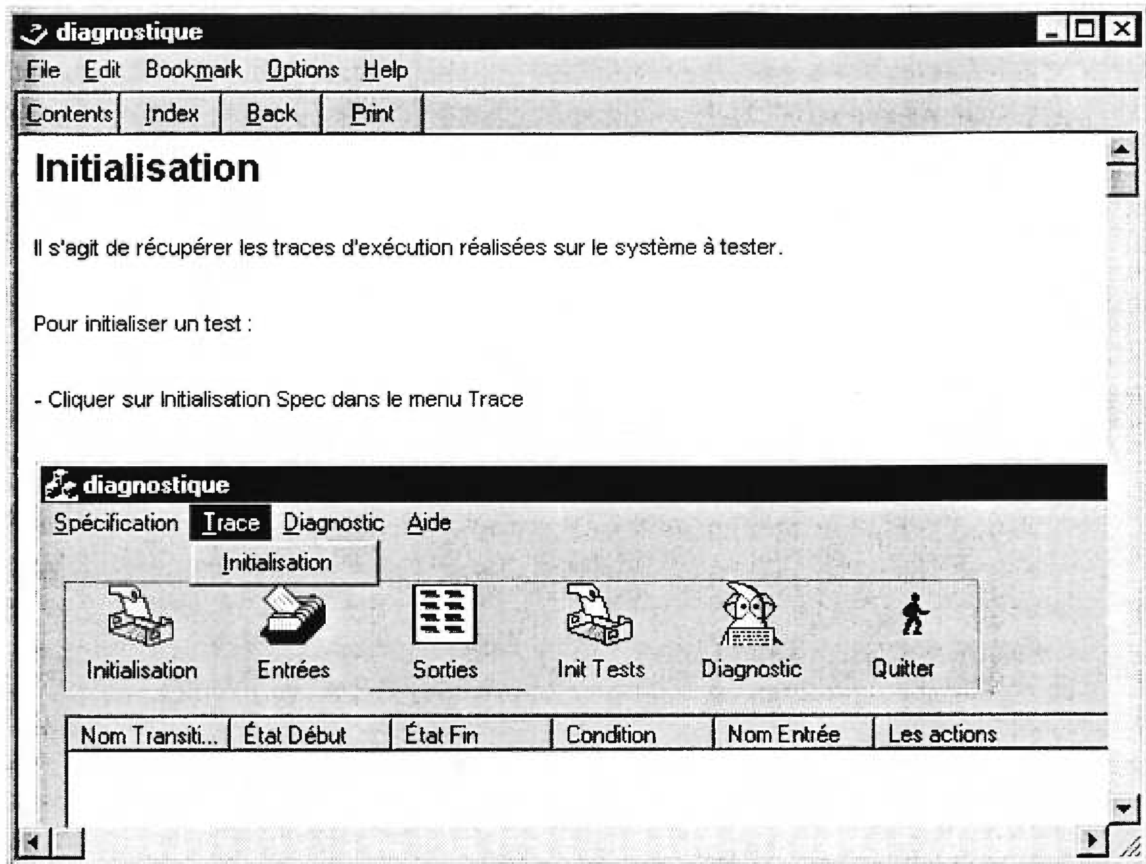


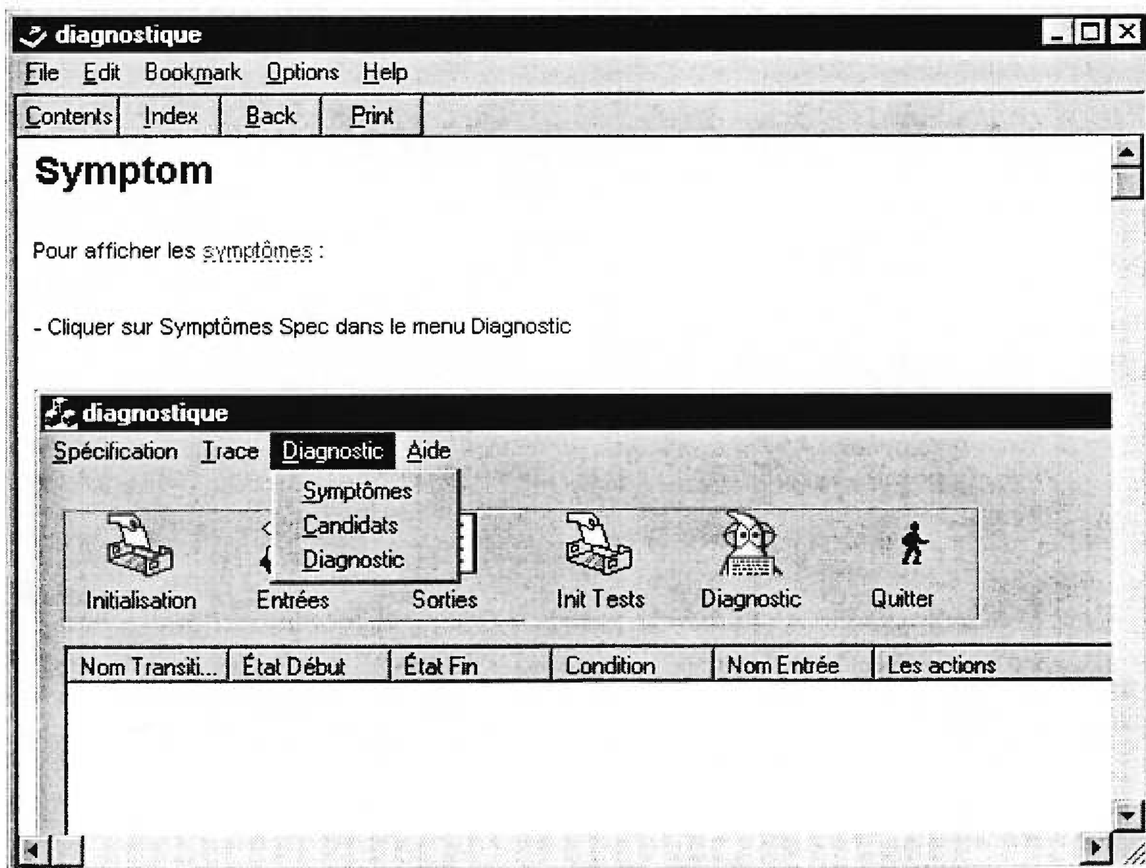

diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Specification Load
Les spécifications sont fournies dans des fichiers textes selon un format spécifique.
Pour charger une specification :
Modèle du système
- Cliquer sur Charger Spec dans le menu Spécification
diagnostique
Spécification Trace Diagnostic Aide
Charger Spec
Initialisation
Entrées
Sorties
Exit
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions

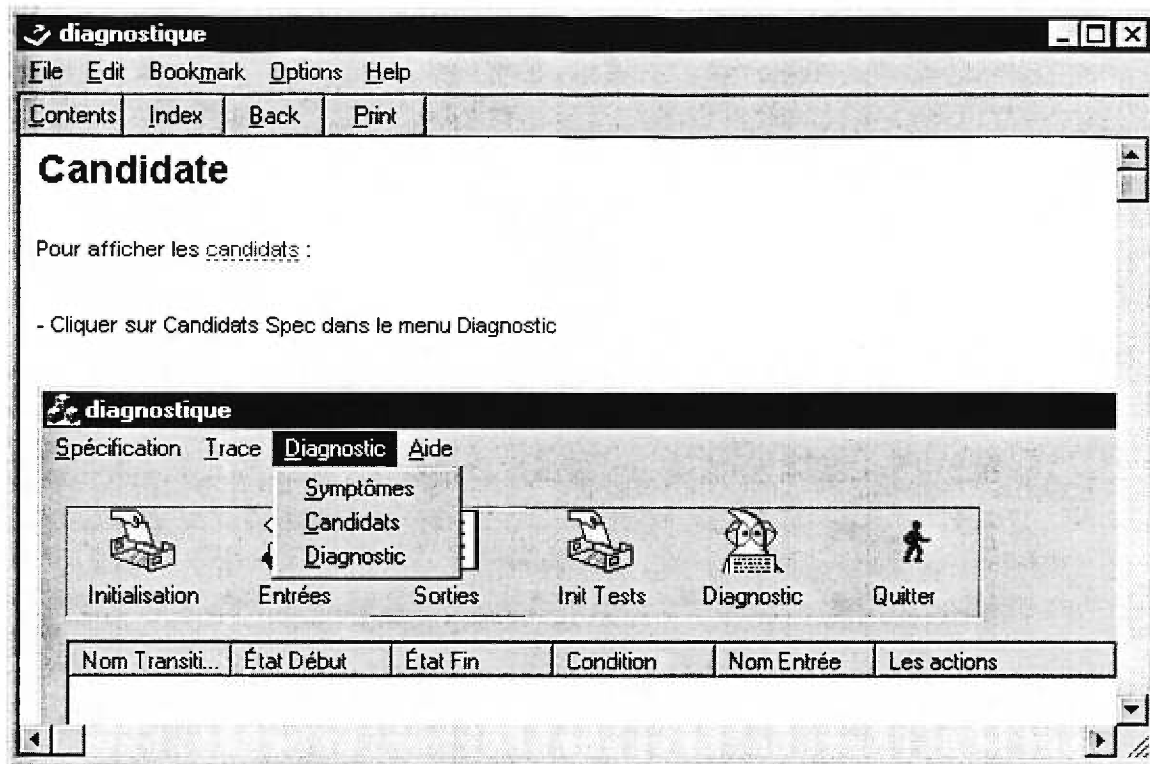

diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Initialisation
Une fois le fichier des spécifications chargé et décortiqué, la spécification est affichée.
Pour initialiser une specification :
- Cliquer sur Initialisation Spec dans le menu Spécification
diagnostique
Spécification Trace Diagnostic Aide
Charger Spec
Initialisation
Entrées
Sorties
Exit
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions

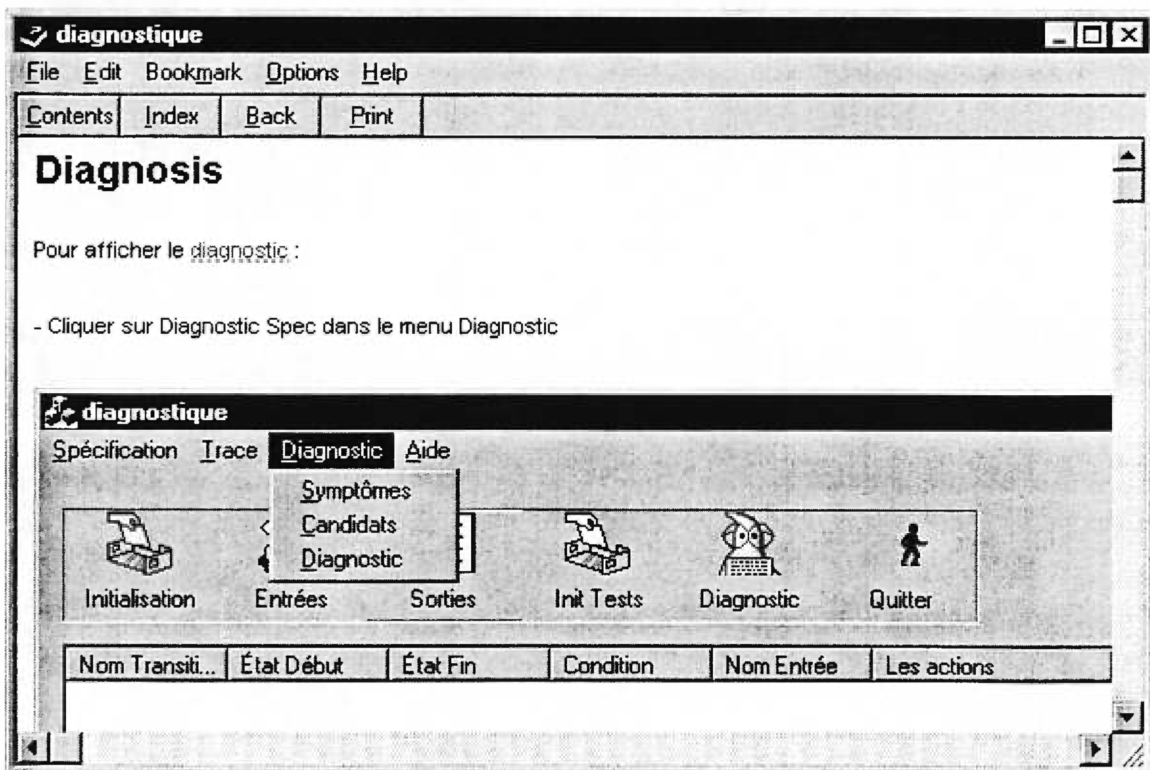

diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Input
Pour lister les entrées de la specification :
- Cliquer sur Entrées dans le menu Spécification
diagnostique
Spécification Trace Diagnostic Aide
Charger Spec
Initialisation
Entrées
Sorties
Exit
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti...
État Début
État Fin
Condition
Nom Entrée
Les actions

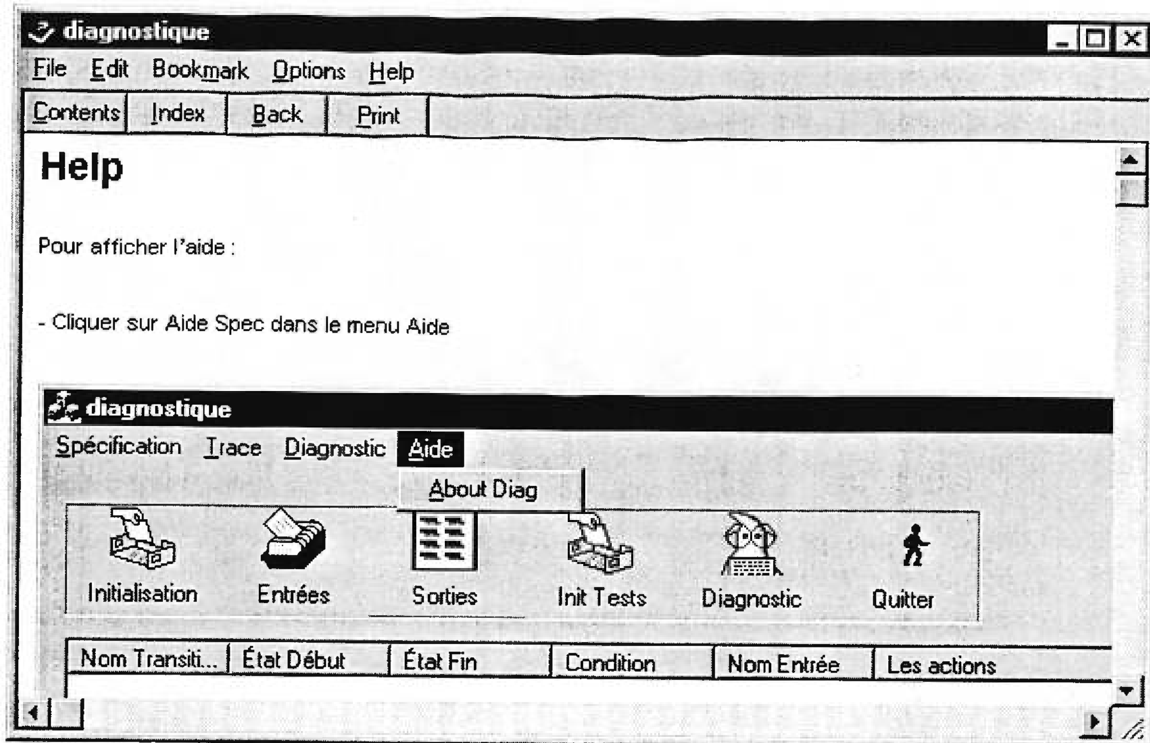

diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Output
Pour lister les sorties attendues de la specification :
- Cliquer sur Sorties dans le menu Spécification
diagnostique
Spécification Trace Diagnostic Aide
Charger Spec
Initialisation
Entrées
Sorties
Exit
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti...
État Début
État Fin
Condition
Nom Entrée
Les actions

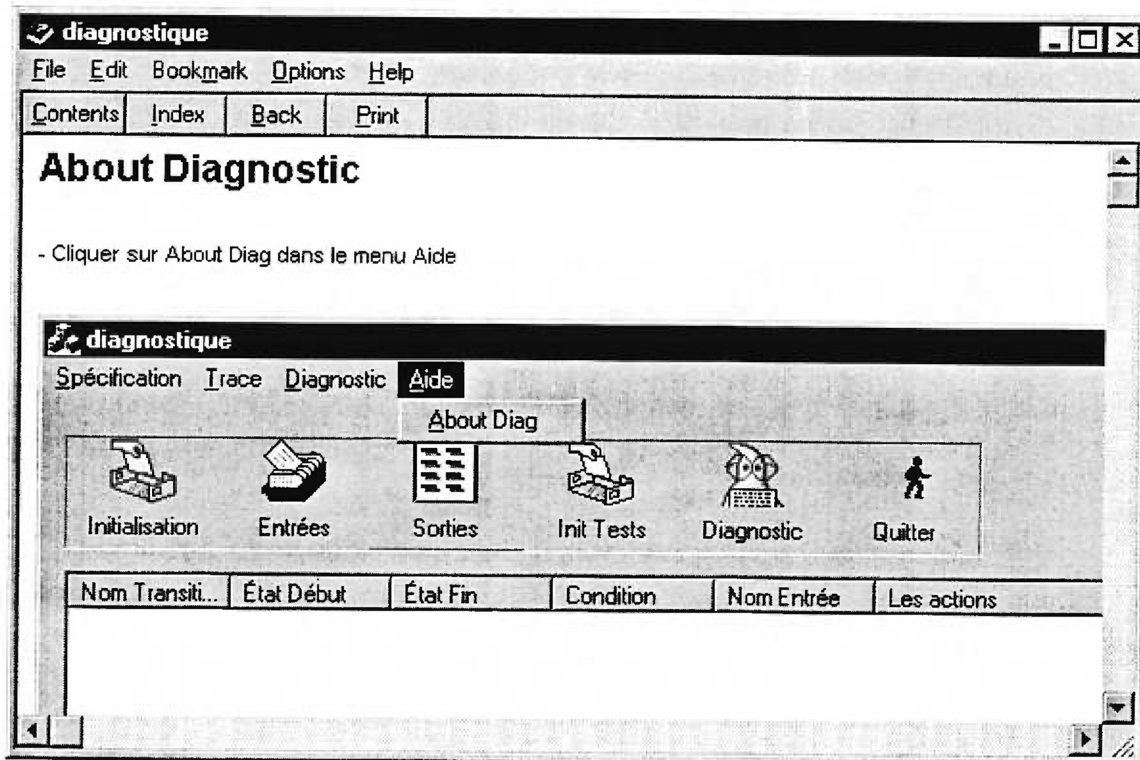

diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Initialisation
Il s'agit de récupérer les traces d'exécution réalisées sur le système à tester.
Pour initialiser un test :
- Cliquer sur Initialisation Spec dans le menu Trace
diagnostique
Spécification Trace Diagnostic Aide
Initialisation
Initialisation
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti...
État Début
État Fin
Condition
Nom Entrée
Les actions


diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Symptom
Pour afficher les symptômes :
- Cliquer sur Symptômes Spec dans le menu Diagnostic
diagnostique
Spécification Trace Diagnostic Aide
Symptômes
Candidats
Diagnostic
Initialisation
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti...
État Début
État Fin
Condition
Nom Entrée
Les actions


diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Candidate
Pour afficher les candidats :
- Cliquer sur Candidats Spec dans le menu Diagnostic
diagnostique
Spécification Trace Diagnostic Aide
Symptômes
Candidats
Diagnostic
Initialisation
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions


diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Diagnosis
Pour afficher le diagnostic :
- Cliquer sur Diagnostic Spec dans le menu Diagnostic
diagnostique
Spécification Trace Diagnostic Aide
Symptômes
Candidats
Diagnostic
Initialisation
Entrées
Sorties
Init Tests
Diagnostic
Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions


diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
Help
Pour afficher l'aide :
- Cliquer sur Aide Spec dans le menu Aide
diagnostique
Spécification Trace Diagnostic Aide
About Diag
Initialisation Entrées Sorties Init Tests Diagnostic Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions


diagnostique
File Edit Bookmark Options Help
Contents Index Back Print
About Diagnostic
- Cliquer sur About Diag dans le menu Aide
diagnostique
Spécification Trace Diagnostic Aide
About Diag
Initialisation Entrées Sorties Init Tests Diagnostic Quitter
Nom Transiti... État Début État Fin Condition Nom Entrée Les actions