Critères de couverture structurelle pour les programmes Lustre

Abdesselam Lakehal

Université Joseph Fourier - Grenoble 1

Critères de Couverture Structurelle pour les Programmes Lustre

Thèse

présentée par

Abdesselam LAKEHAL

en vue de l'obtention du grade de

Docteur de l'Université Joseph Fourier Grenoble 1

Discipline Informatique

soutenue le 08 Septembre 2006 devant le jury composé de :

M. Jean-Claude Fernandez	Professeur Univ. Joseph Fourier	**Président**
M. Bruno Legeard	Professeur Univ. Franche Comté	**Rapporteur**
M. Michel Rueher	Professeur Univ. Sophia-Antipolis	**Rapporteur**
Mme. Virginie Wiels	Chercheur Onera/Cert	**Examinateur**
M. Farid Ouabdesselam	Professeur Univ. Joseph Fourier	**Examinateur**
M. Ioannis Parissis	Maître de Conférences UJF	**Examinateur**

Critères de Couverture Structurelle pour les Programmes LUSTRE

Résumé

Ce travail porte sur le test structurel des programmes réactifs synchrones érits en LUSTRE et sur la définition de critères de couverture pour assurer ce test structurel. LUSTRE est un langage réactif synchrone flot de données, largement utilisé pour la spécification et la programmation des applications critiques dans des domaines comme l'aéronautique, l'énergie ou les transports. L'application à LUSTRE des critères de couverture usuels basés sur le flot de contrôle (couverture des instructions, couverture des branches,...) n'est pas significative.

Nous avons, en conséquence, défini une hiérarchie de critères adaptés au paradigme flot de données synchrone. Les critères permettent de mesurer la couverture des chemins dans un réseau d'opérateurs. Un réseau d'opérateurs est une modélisation graphique des flots de données dans un programme LUSTRE. Les critères sont basés sur le calcul symbolique des conditions d'activation de ces chemins.

Un outil nommé LUSTRUCTU, a été réalisé pour mesurer la couverture atteinte par un jeu de tests selon ces critères. LUSTRUCTU analyse le programme sous test et calcule les chemins et leurs conditions d'activation. L'évaluation de la couverture par LUSTRUCTU est non-intrusive (pas d'instrumentation du code). L'applicabilité et la pertinence des critères ont été évaluées sur une étude de cas significative issue du secteur de l'aéronautique.

Structural Coverage Criteria for Lustre Programs

Abstract

This work deals with the structural testing of the synchronous reactive programs written in Lustre and the definition of criteria to ensure the coverage of this structural testing. Lustre is a data-flow reactive synchronous language, widely used for the specification and the programming of the critical applications in fields like aeronautics, energy or transport. The application to Lustre of the usual control-flow based coverage criteria (statement coverage, branch coverage...) is not significant.

We, consequently, defined a hierarchy of criteria tailored to the data-flow synchronous paradigm. The criteria make it possible to measure the coverage of the paths in an operator network, a graphical model of the data flows in a Lustre program. The criteria are based on a symbolic computation of the activation conditions of these paths.

The criteria were implemented in a tool, Lustructu, which is able to measure the coverage reached by a test set according to these criteria. Lustructu analyzes the program under test and computes the paths and the associated activation conditions. The evaluation of the coverage using Lustructu is not-intrusive (no instrumentation of the code). The applicability and the relevance of the criteria were evaluated on an important case study from the avionics field.

Remerciements

Le travail réalisé dans le cadre de cette thèse a été effectué au sein de l'équipe VASCO du laboratoire LSR-IMAG à Grenoble. Je tiens à exprimer toute ma reconnaissance à son directeur M. FARID OUABDESSELAM et à l'ensemble de l'équipe administrative en particulier PASCALE, MARTINE et LILIANE pour l'accueil et les conditions de travail exceptionnelles.

Je tiens, également, à exprimer mes sincères remerciements à :

MM. MICHEL RUEHER, Professeur à l'université de Nice et BRUNO LEGEARD, Professeur à l'université de Besançon pour avoir accepté d'être les rapporteurs de ce travail.

M. JEAN-CLAUDE FERNANDEZ, Professeur à l'université Joseph Fourier qui m'a fait l'honneur de présider ce jury et Mme VIRGINIE WIELS, chercheur Onera (Toulouse) qui a bien voulu accepter de participer au jury.

MM. FARID OUABDESSELAM et IOANNIS PARISSIS respectivement Professeur et Maître de Conférences à l'Université Joseph Fourier pour m'avoir dirigé, guidé, conseillé ainsi que pour m'avoir fait preuve de tant de confiance, de patience, d'amitié et de bienveillance.

Mme CHRISTEL SEGUIN (Onera), Mme LYDIE DU-BOUSQUET et M. ROLAND GROZ (VASCO) pour leurs conseils qui ont alimenté les réflexions de ce travail.

MOURAD, BESNIK, BRAHIM, JERÔME et LAYA pour leur amitié et pour toutes les discussions que nous avons eues durant ces années de thèse et le stagiaire TONY BERNARD qui a participé à ce travail.

Mes parents, toute ma famille et OUAHIBA pour leur soutien continu et toute leur affection.

à mes parents...

à toute ma famille...

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Cette thèse porte sur le test structurel des programmes réactifs synchrones écrits en LUSTRE et sur la définition de critères de couverture pour assurer ce test structurel.

1.1 Contexte de recherche

Nous nous intéressons aux programmes écrits dans des langages dits réactifs synchrones [4, 5]. Un programme réactif est qualifié de synchrone si sa réaction à ses entrées est instantanée ou, d'une manière plus pratique, s'il est assez rapide pour prendre en compte toute variation significative de son environnement. Cette hypothèse, dite de *synchronisme*, caractérise l'approche synchrone qui a connu un grand essor depuis plus d'une vingtaine d'années, en particulier grâce à des langages dédiés comme ESTEREL [4], SIGNAL [32] et LUSTRE [10, 19]. Ces langages réactifs synchrones sont particulièrement bien adaptés à la description des logiciels dans les systèmes du type contrôle/commande où le logiciel doit agir sur son environnement pour éviter tout dysfonctionnement susceptible de provoquer des catastrophes. Ce type de logiciels est utilisé dans les parties critiques des applications dans des domaines comme l'aéronautique, l'énergie et les transports.

LUSTRE [10] est un langage déclaratif à flots de données synchrones munis d'opérateurs temporels permettant de se référer à des valeurs passées des données. Il a

été conçu pour la spécification et la programmation des logiciels réactifs synchrones dont le comportement est cyclique. Un flot de données est une suite finie ou infinie de valeurs associées à des instants discrets d'une horloge globale. Un programme LUSTRE décrit des fonctions entre des flots en sortie et des flots en entrée. Structurellement, un programme LUSTRE ressemble à un ensemble de blocs échangeant des données sous forme de flots. Les réseaux d'opérateurs sont un moyen graphique usuel pour décrire une telle structure.

Un réseau d'opérateurs est un graphe orienté où les noeuds dénotent des fonctions (appelées opérateurs) et où les arcs dénotent les flots de données entre les noeuds. Un chemin du graphe dénote des dépendances entre les données qu'induisent les traitements au sein d'un programme LUSTRE. La longueur d'un chemin est définie comme le nombre d'arcs qui le composent. SCADE [1] est un environnement de développement dérivé du langage LUSTRE utilisant les réseaux d'opérateurs comme notation. Il comprend une suite composée de nombreux outils de génération de code, de simulation graphique et de vérification. Il est largement utilisé pour le développement des systèmes de contrôle de vol dans des grands projets (A340-600, A380,....), chez le constructeur aéronautique européen AIRBUS et devient de-facto une norme dans ce domaine.

L'utilisation du langage LUSTRE et de son environnement SCADE dans des applications nécessitant un niveau élevé de criticité fait que les activités de vérification et de validation revêtent un intérêt particulier dans le processus de développement. Les phases de vérification et validation sont donc très développées et leur outillage est indispensable. Elles ont été abordées dans le passé principalement au moyen de la preuve formelle basée sur le model-checking [20] ainsi que par le biais du test logiciel en particulier bien fondé mathématiquement [15, 33, 44, 46, 52].

Ce travail s'inscrit, donc, dans un contexte caractérisé par une forte demande de techniques formelles pour le test des applications réactives synchrones. Dans ce cas, il a pour but d'aider à la définition de critères d'évaluation de la qualité des tests, en particulier sous la forme de critères d'arrêt. Par ailleurs, il complète les approches formelles en offrant un moyen de validation/vérification dans des contextes faisant

moins appel à des techniques formelles.

1.2 Problèmes et motivations

De façon générale, le **test logiciel** est un moyen dynamique de vérification et de validation des programmes. Son objectif le plus courant est de révéler des défauts (fautes) dans ces programmes en les exécutant sur des données appelées données de test [40]. Il consiste en un processus en quatre phases : *i)* une phase de *sélection* pour choisir les données de test appelées aussi jeux de tests, *ii)* une phase d'*exécution* pour exécuter le programme sur les données sélectionnées, *iii)* une phase de *verdict* pour comparer le résultat obtenu avec le résultat attendu et enfin *iv)* une phase d'*évaluation* pour s'assurer que le programme a été suffisamment testé.

Deux des problématiques importantes du test logiciel sont relatives aux phases de sélection et d'évaluation. Dans la phase de sélection, le problème est celui du choix des données de test. Idéalement, un test "parfait" est un test exhaustif qui consiste à exécuter le programme sur toutes les valeurs possibles de ses entrées. Cependant, procéder à un test exhaustif est généralement impossible car le domaine de définition des entrées est souvent très large voire infini. L'autre problématique est celle de l'évaluation de la "*qualité*" ou de la "*minutie*" des jeux de tests, qui permet, entre autres, de décider de l'arrêt du processus de test. Des solutions à ces deux problèmes ont été recherchées à travers la définition de règles, qui forment ce qu'on appelle communément critère.

Un **critère** est un ensemble de règles définies sous certaines hypothèses et qui décrivent les propriétés qu'un jeu de tests doit satisfaire. Un critère est dit *critère de sélection* si ces règles servent à construire (à priori) le jeu de tests pendant la phase de sélection. Il est appelé *critère* d'*arrêt* ou d'*adéquation* si les règles sont utilisées comme moyen de s'assurer que le programme sous test a été suffisamment testé en utilisant le jeu de tests. Habituellement, ces critères sont définis comme des mesures de couverture sur le programme lui-même, sur sa spécification ou sur d'autres modèles (fautes, erreurs,...).

Pour les programmes écrits dans des langages impératifs (C, JAVA, ADA,...), une variété de critères de couverture basés sur l'exploitation de la structure des programmes a été proposée. Définis sur le graphe de flot de contrôle (une représentation graphique du flot de contrôle dans le programme), ces critères visent à analyser soit le flot de contrôle soit le flot de données. La couverture des instructions et la couverture des branches, largement utilisées en milieu industriel [3], mesurent respectivement le pourcentage d'instructions ou de branches du programme qui ont été exécutées au moins une fois durant le test. La couverture des chemins vise à s'assurer que chaque chemin du graphe de contrôle est exécuté au moins une fois. Ce critère est inaccessible dans la quasi-totalité des cas, car en présence de boucles le nombre de chemins peut devenir infini.

D'autres critères, tels que la couverture des chemins de longueur k et la couverture des LCSAJ (*Linear Code Sequence and Jump*) [62], ont comme principale utilité d'offrir des paliers mesurables entre la couverture des branches et celle des chemins. Par ailleurs, des critères à base de l'analyse de flot de données [11,51] ont été proposés. Ces derniers visent la couverture des sous chemins du graphe de contrôle reliant les instructions d'affectation de variables à certaines de leurs utilisations.

Dans le cas du test logiciel des programmes LUSTRE, de nombreuses investigations ont été conduites sur la génération des jeux de tests. Certaines des techniques de génération s'appuient sur une spécification fonctionnelle du programme sous test. Dans LUTESS [15,44,46,52], les données de test sont construites dynamiquement de manière aléatoire parmi toutes celles qui respectent les contraintes de l'environnement du logiciel spécifié en LUSTRE. La génération est effectuée soit de façon équiprobable [44,46,52], ou guidée par des stratégies (profils opérationnels [15], schémas comportementaux [64] ou propriétés de sûreté [57]).

D'autres techniques de génération s'appuient sur la description du programme LUSTRE lui-même. Dans GATEL [33], la génération repose sur une interprétation des constructions du langage LUSTRE par des contraintes sur des variables booléennes ou à valeurs dans des intervalles d'entiers. La génération des données de test consiste à résoudre ces contraintes à l'aide de la programmation logique par contraintes.

Cependant, peu de travaux ont été consacrés au critère d'arrêt et à la couverture des programmes LUSTRE [35, 46]. Dans certaines applications du domaine de l'avionique, la certification des composants logiciels nécessite de se conformer à des normes strictes (norme DO-178B) en matière de couverture du code C (généré à partir des spécifications fonctionnelles LUSTRE/SCADE). Pour un concepteur au niveau LUSTRE/SCADE (spécification), la couverture mesurée sur le code C n'est pas significative pour deux raisons :

1. La compilation LUSTRE-C n'est pas standardisée, ce qui rend difficile toute correspondance entre les parties couvertes du code C et celles du code LUSTRE (deux niveaux d'abstraction distincts).
2. Il est difficile aux utilisateurs LUSTRE/SCADE de relier la couverture du code C à toute notion de couverture des spécifications fonctionnelles de LUSTRE (couverture d'une propriété par exemple)

La solution consiste, donc, à définir des critères de couverture pour le code LUSTRE/SCADE. L'application directe au code LUSTRE des critères de couverture des langages impératifs (graphe de flot de contrôle) n'est pas pertinente. En effet, la nature "*flot de données*" de ce langage fait qu'il existe un graphe de contrôle (souvent très rudimentaire) pour chaque équation (i.e. instruction) du programme. La couverture des branches de cet ensemble de graphes de contrôle s'effectue généralement en quelques pas de test et n'est pas considérée comme un critère pertinent pour l'arrêt du test.

La notion de "couverture" des programmes écrits en LUSTRE a déjà fait l'objet de quelques investigations. Dans [35], une mesure de la couverture des comportements des programmes LUSTRE a été proposée. Le test statistique y a été utilisé pour la génération de données assurant la couverture des états et des transitions de l'automate produit par le compilateur LUSTRE. Or, cet automate est souvent de taille et de complexité prohibitives. Une autre approche de couverture définie sur le code LUSTRE [46] a consisté à proposer des critères sur les éléments de base du réseau d'opérateurs (couverture des arcs et couverture des opérateurs). Cependant, ces critères sont souvent satisfaits en quelques pas de test et ne fournissent pas des

mesures puissantes de la couverture du code LUSTRE.

Le but de ce travail, donc, est de proposer des critères de couverture adaptés au langage LUSTRE. Ces critères doivent permettre de :

- tenir compte des spécificités du paradigme synchrone flot de données (dimension temporelle, dépendances de données,...)
- offrir des moyens mesurables, et toujours utilisables, pour évaluer la couverture du code LUSTRE.
- fournir, à terme, des moyens pour décider de l'arrêt de test.

1.3 Contributions de la thèse

A travers ce travail, notre contribution est double *théorique* et *pratique*. Sur le plan théorique, nous proposons une hiérarchie de critères de couverture adaptés aux langages synchrones à flot de données (LUSTRE), ainsi qu'un cadre méthodologique pour l'application de ces critères. Sur le plan pratique, nous avons effectué un certain nombre de réalisations (développement d'un outil de mesure LUSTRUCTU) et d'expérimentations (étude de cas dans les domaine de l'avionique) afin de montrer l'applicabilité et la pertinence des critères en termes de détection de fautes.

Couverture du réseau d'opérateurs

Les premiers travaux sur la couverture des réseaux d'opérateurs [44,46] ont donné lieu à la définition de critères pour la couverture des éléments de base d'un réseau d'opérateurs, à savoir la couverture de tous les arcs (données) et la couverture de tous les opérateurs (fonctions). Ces deux critères fournissent deux moyens de mesure de la couverture du réseau d'opérateurs simples mais insuffisants. Un troisième critère plus fort, la couverture de tous les chemins du réseau, a été proposé. Cependant, ce critère est difficilement satisfiable car le nombre de chemins est souvent infini.

Au lieu de couvrir des éléments de base (arcs et opérateurs) qui composent le réseau d'opérateurs comme le font les critères définis dans [44, 46], les critères que nous proposons sont définis sur des chemins qui ont une longueur quelconque mais

finie. La définition de critères de test en termes de couverture de certains chemins du réseau d'opérateurs s'identifie à celle des fonctions significatives parmi les fonctions réalisées par les chemins.

Formalisation des notions de chemin et de son activation

Un programme LUSTRE réalise une fonction associant à une séquence d'entrées une séquence de sorties. Il s'agit en réalité d'un vecteur de fonctions dont chacune calcule la valeur d'une seule variable de sortie et exprime ainsi l'ensemble de dépendances entre cette variable de sortie et les variables d'entrée. Ce sont ces dépendances que nous essayons de mettre en évidence en définissant des chemins dans le réseau d'opérateurs. Elles peuvent être assimilées à des fonctions partielles à une seule variable intervenant dans le calcul de la sortie. Pour exprimer ces dépendances, nous faisons associer à chaque chemin une notion d'activation que nous formalisons par des expressions booléennes temporelles.

A la différence des critères basés sur le flot de contrôle, la couverture d'un chemin dans un réseau d'opérateurs n'est pas exclusive avec la couverture des autres chemins. En effet, comme un chemin est une fonction partielle, sa couverture entraîne souvent la couverture d'autres chemins faisant partie de la même fonction. En exigeant la couverture de tous les sous-chemins de longueur différente, les critères fournissent un moyen rudimentaire de forcer le test de toutes les fonctions partielles réalisées par les chemins (ou au moins de tenter de le faire). Ce type de test est proche des techniques appliquées au test des circuits digitaux. Cependant, les fautes recherchées dans ce dernier cas sont clairement identifiées (collages de certains fils).

Critères progressifs

Un bon critère de couverture est un critère qui offre un bon compromis entre l'efficacité de détection des fautes et le coût du test. Dans les programmes impératifs, un tel compromis est recherché à travers les critères basés sur les métriques TER_i. Ces métriques définissent une approche incrémentale, par paliers progressifs, pour

mesurer la couverture d'un graphe de flot de contrôle. Comme la couverture de tous les chemins (100%) est impossible dans les programmes, l'idée étant de couvrir des éléments du graphe dont le nombre est toujours fini. Ces éléments sont soit des instructions, des conditions, des séquences de code sans branchements,...etc. La couverture de toutes les instructions (TER_1), la couverture de toutes les branches (TER_2) et la couverture de tous les LCSAJ (TER_3) sont des exemples de critères permettant de mettre en oeuvre une telle approche.

De manière similaire, la couverture de tous les chemins est impossible à atteindre dans les réseaux d'opérateurs comprenant des cycles pour des séquences dont la longueur n'est pas limitée. L'idée, donc, est d'introduire une notion de couverture progressive basée sur des sous-chemins du réseau d'opérateurs dont la longueur est toujours finie. Nous proposons une hiérarchie de critères de couverture applicables aux réseaux d'opérateurs. Cette hiérarchie est basée d'une part sur une construction incrémentale des chemins selon leur longueur et d'autre part sur un renforcement de chaque critère selon sa force.

Longueur des chemins

Un chemin de longueur n est obtenu par la concaténation d'un chemin de longueur $(n-1)$ et d'un chemin unitaire (longueur 2). L'idée est de construire des chemins de plus en plus longs en partant des chemins unitaires. La construction est finie si le chemin est sans cycles ou si le nombre de cycles qu'il contient est fini. En limitant le nombre de cycles dans un chemin, on obtient un chemin de longueur finie, et par conséquent un nombre fini de chemins dans le réseau.

La première orientation consiste donc à associer un critère à chaque classe de chemins de longueurs finies, qu'on identifie par un ordre. On appelle une classe de chemins d'ordre i l'ensemble de chemins dont la longueur est inférieure ou égale à i.

Force intrinsèque du critère

La seconde orientation suggère de définir plusieurs niveaux de force pour un critère d'un ordre quelconque. L'idée est de partir d'un critère de base satisfaisant des

contraintes minimales et de le renforcer en rajoutant progressivement des contraintes supplémentaires de manière à rendre le critère plus contraint. Trois critères sont définis.

Pour le critère le plus simple appelé critère de *couverture de base*, les contraintes minimales portent sur les conditions d'activation des chemins qu'un jeu de tests doit satisfaire. Intuitivement, la satisfaction de ce critère pour un chemin quelconque doit permettre de propager sa valeur d'entrée jusqu'à sa sortie, et ce sans faire d'hypothèses sur cette valeur.

Le deuxième critère, appelé critère de *couverture des conditions élémentaires* doit permettre d'obtenir une forme de dépendance plus forte que la première et par conséquent un critère plus fort. Cette dépendance consiste à forcer la propagation de toutes les valeurs possibles de l'entrée d'un chemin booléen. En d'autres termes, ceci revient à imposer la satisfaction de la condition d'activation du chemin pour toutes les valeurs possibles de l'entrée du chemin ($true$ et $false$).

Le troisième critère est obtenu en généralisant la variation effectuée sur l'entrée du chemin à tous les arcs internes booléens du chemin. Ceci consiste à imposer la satisfaction de la condition d'activation du chemin pour toutes les valeurs possibles de chacun des arcs internes du chemin ($true$ et $false$). Ce troisième critère est appelé *couverture des conditions multiples.*

Sur le plan pratique, les critères de couverture ci-dessus ont été implémentés dans un outil de mesure appelé LUSTRUCTU, dont l'un des principaux objectifs est d'automatiser la mesure de la couverture selon ces critères. L'outil offre, également, un moyen d'expérimenter la couverture selon ces critères sur des études de cas académiques et industrielles ainsi que la possibilité d'être utilisé avec d'autres outils de génération existants pour des fins de mesure de la couverture.

1.4 Organisation du document

Ce document s'articule autour de trois parties :

La **première partie**, composée des chapitres 2 et 3, est consacrée à un état

de l'art de la couverture dans le test logiciel et au test des programmes synchrones écrits en LUSTRE. Le chapitre 2, intitulé *test structurel et couverture*, dresse un état de l'art de la notion de couverture dans le test logiciel. En particulier, ce chapitre met l'accent sur les travaux effectués sur la couverture à base des graphes de flot de contrôle.

Dans le chapitre 3, nous introduisons, en premier, les éléments structurels et comportementaux du langage LUSTRE. Nous présentons, ensuite, les différents travaux effectués sur le test des programmes synchrones durant ces dernières années et en particulier ceux autour de la génération des données de test et les différentes techniques de génération basées sur les spécifications fonctionnelles.

La **deuxième partie** est constituée du chapitre 4 et consacrée à la définition formelle des critères que nous avons proposés. Dans ce chapitre, nous introduisons des définitions formelles des réseaux d'opérateurs, des orientations qui guident la définition des critères ainsi que des définitions des trois familles des critères (le critère de couverture de base, le critère de couverture de conditions élémentaires et le critère de couverture de conditions multiples) avec des exemples d'illustration et des comparaison avec des critères existants. Cette partie décrit, également, une approche de couverture basée sur ces critères.

Dans la **troisième partie** (chapitre 5), nous fournissons une évaluation empirique des critères et une étude comparative des résultats de la couverture obtenue sur les trois critères entre. Nous présentons les résultats effectués sur une étude de cas du monde de l'aéronautique, dans lesquels nous illustrons la pertinence des critères définis au chapitre 4 et leur aptitude à révéler des fautes dans les programmes LUSTRE.

Enfin, des conclusions ainsi que des perspectives et des questions ouvertes de ce travail sont présentées au chapitre 6.

Chapitre 2

Test Structurel et Couverture

2.1 Test logiciel

La sûreté de fonctionnement [30] (la crédibilité selon [49]) désigne la propriété qui permet d'avoir confiance dans le service délivré par un logiciel. Cette propriété peut être obtenue à l'aide de moyens de validation et/ou de vérification qui visent, entre autres, l'élimination des fautes. Une faute (appelée également défaut) est définie selon [30] comme la cause supposée ou adjugée d'une erreur, où celle-ci représente l'état du programme susceptible de provoquer un comportement incorrect (défaillance). De nombreuses techniques ont été proposées pour l'élimination de fautes. Ces techniques vont de la preuve formelle de propriétés au test du logiciel. Dans ce travail, nous nous intéressons à la deuxième technique et en particulier au test pour la correction.

Le **test logiciel** est un moyen dynamique d'élimination de fautes, par opposition aux moyens statiques comme l'inspection du code [40] et les revues [40]. Il consiste à exécuter un programme tout en ayant la totale maîtrise des données qui lui sont fournies en entrée et tout en vérifiant que son comportement est celui attendu. Il sert à la fois pour la vérification que pour la validation des programmes.

Le *test pour la vérification* suppose l'existence d'une spécification explicite et complète et vise à démontrer que l'implémentation y lui est conforme. Par exemple, le test de conformité [55] dans le domaine des protocoles est un moyen de vérification.

Son but est de s'assurer qu'une implantation d'un protocole répond (est conforme) aux exigences formulées dans sa spécification.

Le *test pour la validation*, en revanche, repose sur une spécification partielle voire implicite et vise, tout autant, à provoquer des défaillances qu'à conclure que le programme se comporte bien conformément à ce qu'on attend de lui. Selon [40], *"tester un programme, c'est l'exécuter dans l'intention d'y trouver des anomalies ou des défauts"*. Cette définition est proche de celle de Dijkstra [40] qui considère le test logiciel comme un moyen pour *révéler la présence des fautes mais jamais leur absence.*

L'identification d'un écart entre le comportement du programme et le comportement attendu "*spécification*" est du rôle de l'*oracle*, une notion mise en évidence par la définition suivante (IEEE) "*le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus*". L'usage du terme d'oracle est consécutif à la nécessité de conclure ou non à un écart entre les comportements, en l'absence de spécifications complètes.

2.1.1 Processus de test

L'activité de test est fortement liée aux pratiques des testeurs et aux types de programmes à tester. Néanmoins, il est commun de considérer cette activité comme un processus qui se déroule en quatre étapes :

Sélection

L'étape de sélection consiste à choisir, parmi l'ensemble des valeurs possibles des entrées, un sous-ensemble, appelé *données de test* à soumettre au programme sous test. Plusieurs définitions existent dans la littérature concernant les termes de "*jeu de tests*", "*cas de test*" , "*données de test*" et "*séquence d'entrées*". Dans le cadre de ce travail, nous considérons qu'une *donnée de test* est une instance du produit cartésien des domaines de valeurs d'entrée du programme.

Un *cas de test* (ou tout simplement un *test*) comprend, en plus de la donnée

de test, le résultat attendu après l'exécution du programme. La définition des cas de test suppose l'existence d'une spécification (entrées, sorties et fonction). Enfin, nous appellerons *jeu de tests* (ou tests) un ensemble fini de cas de test. La notion de *séquence d'entrées* sera introduite au chapitre suivant, lorsqu'on aura abordé une classe particulière de programmes qui s'exécutent de manière réactive et continue.

Exécution

Cette étape consiste à soumettre le jeu de tests choisi pendant l'étape de sélection et à exécuter le programme sous test sur chaque cas de test. Généralement, il existe deux modes pour l'exécution de test : un mode en lot (batch) qui consiste à soumettre tout le jeu de tests en vrac et récupérer les résultats ; et un deuxième mode interactif qui consiste à interagir avec le programme à chaque cas de test soumis.

Verdict

Une fois l'exécution du programme sur un cas de test est terminée, le testeur a besoin d'attester de la réussite ou de l'échec du test en comparant les résultats obtenus avec les résultats attendus. L'observation de l'exécution du logiciel doit permettre d'identifier les défaillances. Cette étape suppose l'existence d'un *oracle* qui distingue le comportement erroné du comportement correct.

Évaluation

Cette étape consiste à évaluer la *qualité* du test effectué et permet, entre autres, de décider ou non de l'*arrêt de test*. Dans ce qui suit, nous considérons la qualité du test et la qualité du jeu de tests sous-jacent comme deux propriétés se référant à la même chose. Bien que souvent considérée comme subjective, la qualité d'un jeu test est une notion fortement liée à l'**objectif du test**.

2.1.2 Objectifs du test

Comme nous l'avons souligné plus haut, le test logiciel est un moyen qui permet d'accroître la confiance dans le logiciel. Cette confiance peut être recherchée par :

- l'évaluation de la fiabilité du logiciel
- l'établissement de la conformité avec une spécification.
- la détection des fautes dans le programme.

Dans le cadre de ce travail, nous n'aborderons pas les deux premiers objectifs et nous consacrerons la suite de ce chapitre aux moyens qui permettent ou qui facilitent la détection des fautes dans les programmes. Si l'objectif du test est de révéler des fautes, la qualité des tests peut être caractérisée comme l'aptitude de ces tests à révéler les fautes dans le programme. L'intuition de l'*exhaustivité* stipule que plus le nombre de données de test est grand, plus sa "qualité" augmente. Ainsi avec un test exhaustif (i.e. exécuter le programme sur toutes les valeurs possibles des entrées du programme), le testeur est certain que son test est en quelque sorte "parfait".

Cependant, procéder à un test exhaustif est généralement irréalisable, et ce pour deux raisons principales :

- Le domaine des valeurs des entrées d'un programme, défini comme le produit cartésien des domaines de définition des entrées, peut être très large voire infini. Par exemple, le test exhaustif d'un programme ayant comme entrées trois entiers définis chacun sur 16 bits nécessite l'exécution du programme sur $(2^{16})^3 = 2.814 \text{ x } 10^{14}$ données de test. Pour un compilateur, le domaine d'entrée (ensemble de tous les programmes) est infini.
- Même lorsque le domaine d'entrée est limité, il est possible que le programme contienne une infinité de scénarios d'exécutions possibles. Les programmes impératifs séquentiels comprenant des boucles infinies sont des exemples de tels programmes.

2.1.3 Sélection des données de test

L'un des enjeux du test logiciel est de maximiser la qualité du test avec un jeu de tests réduit, voire minimal. La construction d'un tel ensemble est faite en s'appuyant sur des hypothèses dites *hypothèses de réduction*. Ces hypothèses sont des propriétés qui sont formulées pour réduire l'effort de test et rendre le nombre de données de test fini et abordable. Elles sont construites de manière à garantir la préservation de la pertinence de l'ensemble de données de test construit, et correspondent à une formalisation des pratiques courantes chez les testeurs.

Ces hypothèses se divisent en deux catégories [30] : les hypothèses dites d'*uniformité* et les hypothèses dites de *régularité*. Les hypothèses d'*uniformité* admettent que si un programme se comporte correctement pour un élément d'un ensemble, il se comportera également correctement pour les autres éléments. Les hypothèses de *régularité* stipulent que les comportements d'un programme partageant de grandes séquences d'instructions (de ce programme) sont similaires. Ainsi, il est inutile de tester tous les comportements incluant une même boucle pour révéler tous leurs défauts : retenir seulement quelques comportements correspondant à quelques itérations de cette boucle suffit.

Par ailleurs et en l'absence de modèles caractérisant les fautes d'un programme, d'autres hypothèses permettent d'établir une correspondance entre un ensemble de données de test et les fautes qu'il est susceptible de révéler. De telles hypothèses stipulent, par exemple, que l'exécution de toutes les instructions dans un programme garantit que toutes les fautes relatives aux instructions soient détectées.

L'exigence d'exécution de toutes les instructions dans un programme ou l'obligation de révéler la totalité ou certains types de fautes par un jeu de tests constituent ce qu'on appelle **des critères**.

2.1.4 Critères de sélection versus critères d'adéquation

Un critère est un ensemble de propriétés et de conditions qu'un jeu de tests doit satisfaire afin d'atteindre un objectif. Ces conditions peuvent porter sur le

programme (ou sur un modèle de celui-ci), sur sa spécification ou sur les deux ensembles. Selon que les conditions du critère soient vérifiées pendant la phase de sélection ou pendant la phase d'évaluation, le critère est dit critère de *sélection* ou critère d'*adéquation*.

- Un critère de *sélection* est un moyen de construire, *à priori*, les données de test. Autrement dit, les données de test doivent être choisies de telle sorte qu'elles respectent les conditions stipulées dans le critère.
- Un critère d'*adéquation* est un moyen pour évaluer, *à posteriori*, la "qualité" ou la "minutie" des données de test. Le critère est dit critère d'*arrêt* s'il est utilisé pour décider si un programme a été ou non suffisamment testé.

2.1.5 Comparaison de critères

L'utilisation des relations d'inclusion (voir définition en bas) pour comparer des critères de couverture est largement utilisée dans la plupart des techniques de test. Une relation d'inclusion évalue les critères en leurs propres termes (indépendamment du modèle sur lequel le critère est défini) mais n'exprime rien sur la manière dont les critères révèlent les fautes ni sur leurs capacités à le faire. En se basant sur cette relation d'inclusion, d'autres relations d'ordre total ou partiel formant des hiérarchies de critères peuvent être construites. Nous introduirons au fur et à mesure des exemples de cette relation chaque fois que des critères comparables sont présentés.

Définition 2.1.1 *Soit deux critères C_1 et C_2.*

On dit que C_1 inclut C_2 si et seulement si tous les jeux de tests qui satisfont C_1 satisfont également C_2.

2.2 Méthodes de test et critères

Les méthodes ou techniques de test sont souvent classées en fonction de la manière utilisée pour la sélection des données de test. Pour choisir des données de test, le testeur utilise les documents mis à sa disposition lors de la phase de test ; c'est à

dire : les spécifications, les caractéristiques des entrées et le code du programme. Si la sélection se fait à partir des spécifications, on parle de test fonctionnel (ou test boîte noire), si elle se fait à partir du code de programme, on parle de test structurel (ou test boîte blanche).

2.2.1 Test "boîte noire"

Dans le test boîte noire, les données de test sont sélectionnées soit de manière aléatoire soit à partir des spécifications fonctionnelles du programme. La structure interne (code) du programme est, donc, inconnue ou ignorée. Le testeur considère le programme sous test comme une boîte noire où seules les entrées, les sorties et la fonction du programme sont connues. Le test se déroule en exécutant le programme sur les entrées, et en comparant les résultats de l'exécution avec les résultats attendus, décrits dans la spécification, pour valider sa correction. Il existe plusieurs techniques de test dites "boîte noire". Nous rappellerons les plus connues d'entre elles dans la suite de cette section.

Le ***test aléatoire*** [21] est une technique de test dans laquelle les données de test sont sélectionnées par tirage aléatoire sur le produit cartésien des domaines de définition des entrées du programme sous test. Si les usages futurs du programme sont inconnus, le tirage est effectué de manière équiprobable. Autrement, le tirage est effectué de manière à respecter ces usages dits profils opérationnels. Un profil opérationnel [39] est une caractérisation quantitative de la façon dont le programme est utilisé. Les profils opérationnels définissent une distribution opérationnelle statistique du domaine d'entrée. Selon *Duran et Ntafos* [16], la génération aléatoire équiprobable assure une bonne couverture du code et une bonne détection des fautes. Néanmoins, l'efficacité du test aléatoire peut varier en fonction de l'application et de la distribution à partir de laquelle des données de test sont sélectionnées.

La technique des ***catégories et partitions*** [18] est une autre technique de test "boîte noire", basée sur une décomposition plus fine des domaines de définition des entrées du programme que le test purement aléatoire. Cette technique suppose un partitionnement des domaines de définition des entrées en classes d'équivalence. Le

bien-fondé de cette technique repose sur les deux intuitions suivantes :

1. Chaque donnée de test réduit le nombre de données de test nécessaires d'une unité.
2. Chaque donnée de test est représentative de sa classe d'équivalence dans le sens où elles ont la même aptitude à révéler des fautes.

La technique des ***valeurs aux limites*** est une variante de la technique précédente en considérant non seulement les valeurs à l'intérieur des classes d'équivalence, mais aussi aux limites de chaque classe. Dans cette variante [40], les données de test sont dérivées à partir de la spécification des besoins. Elles sont, ensuite, partitionnées en classes, et les données de test sont sélectionnées à l'intérieur et aux limites de chaque classe.

2.2.2 Test boîte blanche

Contrairement au test boîte noire, les données de test sont sélectionnées dans le but de satisfaire des critères liés uniquement à la structure de code. Le code du programme sous test et ses propriétés (flots de contrôle, flots de données) sont visibles pour le testeur, et le test est conduit selon les détails d'implémentation du programme sous test. Les principales techniques de cette classe sont les techniques basées sur la couverture structurelle et la technique de mutation. Dans ce type de techniques, la nécessité de l'exhaustivité est moins contraignante que dans le test boîte noire, car une certaine exhaustivité peut être atteinte comme l'exécution de toutes les instructions du programme.

Nous consacrerons la suite de ce chapitre à ces différentes techniques et aux critères associés.

2.3 De la couverture dans le test logiciel

Dans le test logiciel, les terme "qualité de test" et "couverture" renvoient souvent à la même chose. Pourtant et de façon générale, le terme couverture ne signifie pas

exactement la même chose selon qu'on s'adresse à un spécialiste du test du matériel ou à un spécialiste du test logiciel. En effet, chacun utilise une mesure de la qualité propre à son métier et que tous appellent la couverture. Le spécialiste du test matériel mesurera la qualité des tests en termes de leur aptitude à détecter les fautes ; on parle alors de couverture de fautes. Le spécialiste du test logiciel exprimera, par exemple, la couverture comme le taux des instructions du programme exécutées.

Le point commun entre ces deux notions est qu'elles s'expriment comme le rapport entre ce qui est atteint par le test (fautes, instructions, ...) et ce qui devrait être idéalement atteint si le test était d'un certain point de vue "parfait". Par conséquent, ces mesures de couverture s'expriment toutes comme des taux ou des pourcentages. Dans la suite de ce chapitre, nous nous intéressons à ces deux types de couverture.

La mesure de la couverture d'un composant logiciel nécessite d'une part l'identification d'un modèle sur lequel la couverture est effectuée et d'autre part l'application de critères de couverture sur les éléments de ce modèle.

Modèle de couverture

Le modèle de couverture d'un programme est l'ensemble des éléments qu'une exécution de ce programme doit mettre en jeu. Ces éléments peuvent être dérivés du composant lui-même (code) ou d'une de ses descriptions graphiques comme le graphe de flot de contrôle ou le graphe de flot de données, ou encore de certaines de ses abstractions. Ils peuvent être également dérivés d'un modèle de fautes du programme s'il y en a un. Nous détaillerons chaque type de modèles dans la section 2.4.

Critère de couverture

Un critère de couverture est une métrique quantitative calculée sur des éléments d'un modèle de couverture. Souvent exprimé en pourcentages, un critère de couverture fournit une mesure sur le nombre d'éléments du modèle mis en jeu par l'exécution du programme sur des données de test. Il permet, sous certaines hypothèses, d'avoir un certain niveau de confiance dans le test. La section 2.5 sera consacrée aux

critères de couverture des graphes.

Dans la suite de ce document, nous nous intéressons plus particulièrement à la couverture des programmes où le modèle de couverture est une représentation fidèle de celui-ci (par exemple le graphe de flot de contrôle pour les programmes impératifs). Ensuite, nous présenterons un état de l'art des critères de couverture définis sur des modèles comme les graphes de flot de contrôle.

2.4 Modèles de couverture

Traditionnellement, il existe deux grandes familles de modèles graphiques pour représenter des programmes : les modèles basés sur le flot de contrôle comme les graphes du type organigramme et les automates et les modèles basés sur le flot de données (structures à blocs comme les réseaux d'opérateurs).

2.4.1 Graphes de flot de contrôle

Dans l'approche à "*flot de contrôle*", l'accent est mis sur le séquencement des opérations (contrôle), et aucune indication n'est faite sur les flux de données. Ce type de graphes impose un ordre total sur les opérations : un seul sommet est activé à la fois. Ce paradigme met en évidence le séquencement des opérations dans un processus, en se focalisant sur l'aspect procédural. Les graphes de flot de contrôle sont utilisés comme des modèles pour décrire la structure des programmes impératifs. Ils servent à la fois pour l'analyse statique [25,36] et comme modèle pour la couverture des programmes [61].

Un **graphe de flot de contrôle** est un graphe orienté, composé d'un ensemble de noeuds reliés par un ensemble d'arcs orientés. Un *arc* (branche) décrit une séquence linéaire contiguë d'opérations (instructions) entre deux noeuds. Un *noeud* dénote un point d'entrée du programme (noeud d'entrée), un point de sortie (noeud de sortie) ou un transfert de contrôle (branchement, noeud interne).

A chaque noeud interne, est associé une expression booléenne représentant la condition du transfert de contrôle. Cette expression peut être soit une *condition*

```
read(x) ;
read(y) ;
signe := 1 ;
z := 0 ;
if (x<0) then
begin
  signe := -1 ;
  x := -x ;
end ;
if (y<0) then
begin
  signe := -signe ;
  y := -y ;
end ;
while (x >= y) do
begin
  x := x-y ;
  z := z+1 ;
end
z := signe*z ;
print(z) ;
```

FIG. 2.1 – Programme de calcul de la division entière

élémentaire, atomique et indivisible (appelée *condition*), soit une condition composée via des connecteurs logiques (appelée *décision*). La distinction entre décision et condition au niveau d'un noeud est utile dans l'analyse d'une expression booléenne que contient le noeud. En effet, l'analyse peut être envisagée à plusieurs niveaux de granularité (condition élémentaire, combinaisons de conditions, ...).

Un chemin du graphe est une séquence d'arcs successifs. Un chemin allant du noeud d'entrée au noeud de sortie correspond à une exécution possible du programme, et appelé *chemin d'exécution*.

Soit un programme simple (voir 2.1) qui calcule le quotient z de la division entière d'un entier x par un entier y. Le graphe de flot de contrôle de ce programme est donné dans la figure 2.2. Dans ce graphe, chaque bloc d'instructions est illustré par une branche (arc) du graphe. Les arcs en pointillés correspondent à des blocs d'instructions vides.

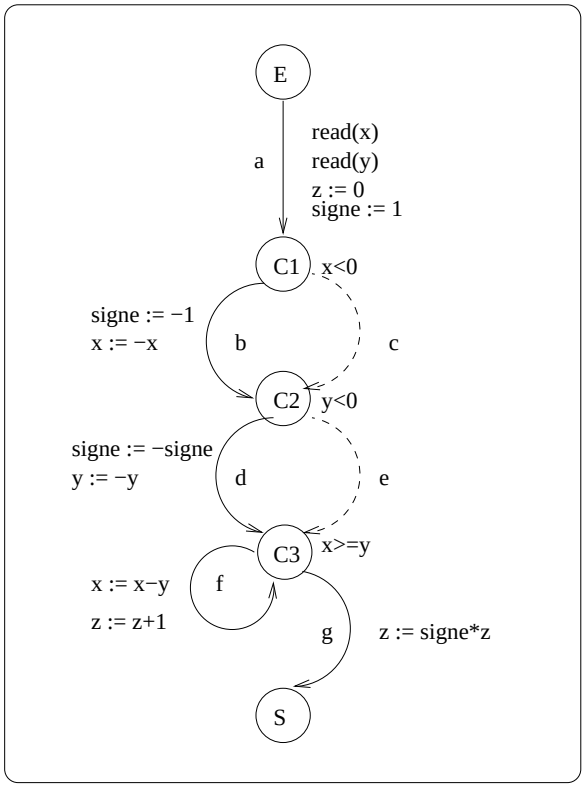

FIG. 2.2 – Graphe de flot de contrôle

2.4.2 Graphes de flot de données

Dans l'approche "*flot de données*", les sommets du graphe représentent des opérations atomiques et les arcs représentent des transferts de données entre ces opérations. Les sommets produisent de façon répétitive des valeurs en sortie en fonction des valeurs reçues en entrée. Des règles de déclenchement spécifient à quel moment un sommet est activé. Le paradigme "flot de données" permet de mettre en évidence les flux d'information dans le programme, et décrit de façon explicite la dépendance entre les données.

Dans cette catégorie de graphes, on trouve plus particulièrement les structures à blocs utilisées pour décrire des systèmes manipulant des flots de données comme les équations différentielles. Dans ce type de graphes, les noeuds représentent des fonctions et les arcs représentent les flots de données entre ces fonctions. Dans un noeud, les arcs entrants représentent les données d'entrée de la fonction (opérandes) et les arcs sortants représentent les données résultant du traitement. L'exécution

de la fonction dans un noeud se produit quand tous ses opérandes (les données en entrée) sont disponibles. Ainsi, les fonctions sont exécutées dans un ordre résultant seulement des dépendances entre les données.

2.4.3 Modèles de fautes

La notion de "modèle de fautes" trouve son origine dans le test de fabrication des circuits intégrés [9]. Dans ce type de test, les fautes probables sont connues (courts-circuits, connexions défectueuses...). Par conséquent, il est réaliste d'envisager la qualité des tests selon leur aptitude à détecter ces fautes. Pour le test logiciel, cette approche s'est concrétisée principalement par la technique de *mutation*. L'idée de la mutation est d'injecter des fautes syntaxiques dans le programme d'origine afin de produire des programmes appelés *mutants*, qui constituent un modèle de fautes.

La construction d'un modèle de fautes est rendue possible grâce aux deux hypothèses du programmeur compétent et celle du couplage fort [12] :

- L'hypothèse du *programmeur compétent* stipule que les programmeurs créent des programmes qui compilent et qui respectent leurs spécifications et que seules des fautes mineures et locales peuvent être commises ; comme l'omission d'une variable, le remplacement d'un opérateur par un autre,....etc.
- L'hypothèse du *couplage fort* suppose que des données de test qui révèlent des fautes mineures dans le programme sont également capables de révéler des fautes résultant du couplage de ces fautes mineures.

La mutation est basée sur des règles appelées *opérateurs* de mutation. Potentiellement, il existe un nombre illimité d'opérateurs mais dans la pratique, ce nombre est souvent restreint [13].

2.4.4 Autres modèles (Spécifications)

Les modèles présentés ci-dessus sont utilisés pour dénoter graphiquement des programmes impératifs. Il existe un autre type de notations graphiques utilisé pour décrire les spécifications. Bien que la couverture des spécifications sort du cadre de ce

travail, nous donnerons néanmoins les modèles les plus utilisés pour la couverture des spécification car certaines similitudes existent entre certains critères de couverture de spécifications et des critères de couverture des programmes.

Graphes causes-effets

Un graphe causes-effets [40] est une notation graphique utilisée pour décrire la spécification d'un logiciel en termes de relations logiques entre des causes et des effets. Une *cause* est une expression atomique booléenne dans la spécification et représente une valeur d'entrée (ou encore un état ou un événement). Un *effet* est une expression atomique booléenne représentant l'état d'une sortie ou une action.

La représentation graphique est dérivée en connectant les causes et les effets au moyen des opérateurs logiques usuels : la conjonction ($\wedge$), la disjonction ($\vee$) et la négation ($\sim$), dont les notations sont illustrées sur l'exemple 2.4.1 (voir Fig.2.3). Le même exemple est décrit graphiquement par un diagramme logique équivalent (voir Fig. 2.4).

Exemple 2.4.1 Considérons la spécification informelle suivante décrivant le fonctionnement d'une chaudière [53]. Cette dernière doit être arrêtée suite à l'occurrence de l'une des conditions suivantes :

1. Le niveau d'eau est au-dessous du niveau de 20.000 (a)
2. Le niveau d'eau est au-dessus de 120.000 (b)
3. La pompe d'eau est défectueuse (c)
4. Le moniteur de la pompe est défectueux (d)
5. Le capteur de vapeur est défectueux (e)
6. La chaudière est mise hors tension (f)

Les phrases de 1 à 5, qui peuvent être vraies ou fausses sont des causes. Certaines de ces combinaisons peuvent conduire à l'effet que la chaudière est mise hors tension. On associe une variable booléenne à chaque cause. Les variables booléennes de a à e constituent des causes dans le graphe. On peut représenter l'effet f par une

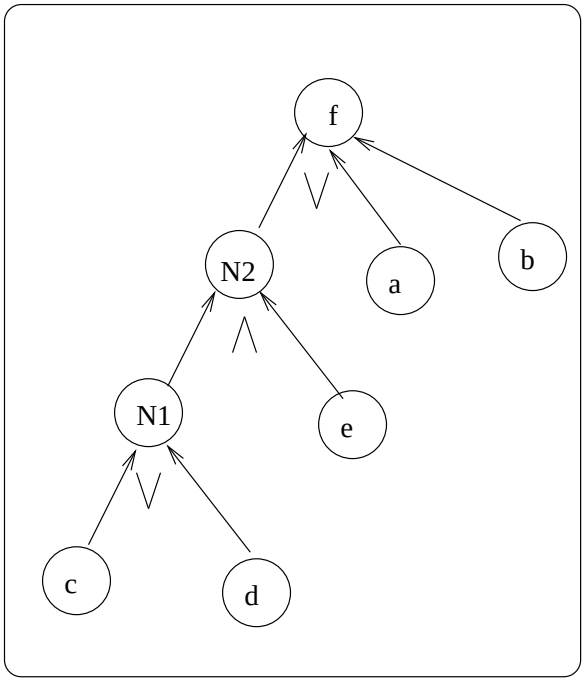

FIG. 2.3 – Exemple d'un graphe causes-effets

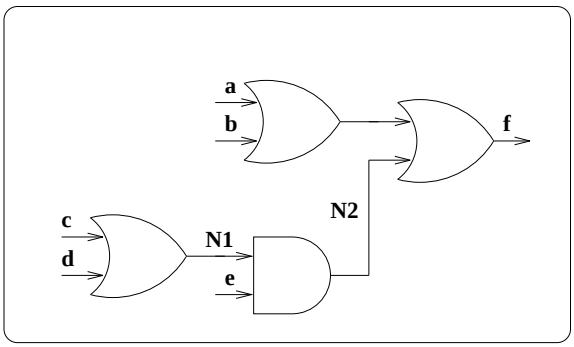

FIG. 2.4 – Représentation sous forme de circuit logique d'un graphe causes-effets

expression booléenne $E = a + b + (c + d)e$. La représentation graphique de cet effet est donnée dans la figure 2.3.

Les graphes causes-effets sont utilisés pour la formalisation et l'expression des besoins [40] ainsi que dans le test logiciel pour dériver des jeux de tests à partir des spécifications [45, 53].

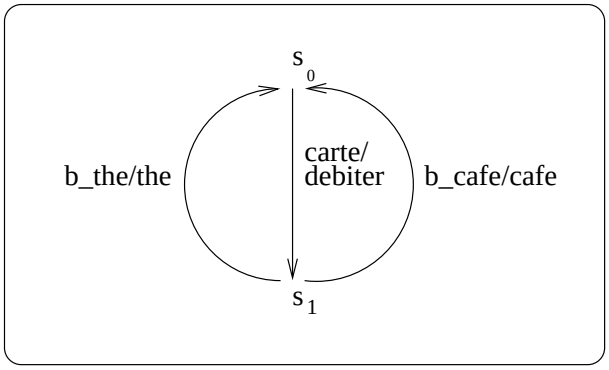

FIG. 2.5 – FSM modélisant le distributeur

Machines d'états finis

Les machines d'états finis (ou machines de Mealy) [24] conviennent parfaitement pour modéliser les systèmes réactifs. Ce genre de modèles est souvent représenté par un graphe dont les noeuds sont des états et les arcs sont des transitions entre états. De plus, les arcs sont étiquetés par l'entrée qui provoque la transition et la sortie qui en résulte. La figure 2.5 illustre une machine à états finis associée à la spécification d'un distributeur de boissons, dont le comportement est décrit informellement comme suit :

1. *On peut obtenir une boisson lorsque la machine n'est pas déjà en état de servir quelqu'un.*
2. *Pour obtenir une boisson (thé ou café), le client doit introduire sa carte d'unités. La carte est alors débitée d'une unité et il peut choisir sur le panneau de commandes entre un thé ou un café.*
3. *Une fois que le bouton est pressé, la boisson correspondante est servie.*
4. *Le distributeur est alors prêt à servir une autre boisson.*

Systèmes de transitions étiquetés

Les systèmes de transitions étiquetés (LTS) ont été largement utilisés pour décrire la sémantique d'un certain nombre de langages de spécification comme CCS [37], CSP [23] ou pour modéliser des implantations particulières [56]. Contrairement aux

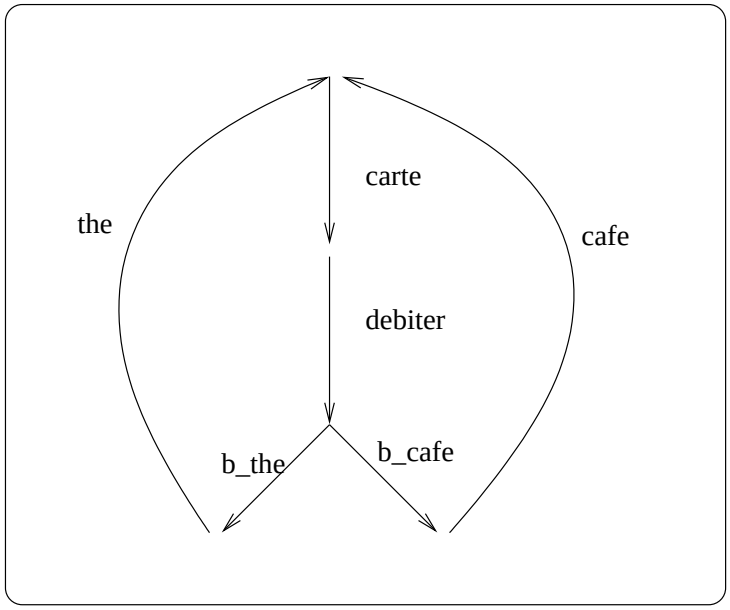

FIG. 2.6 – LTS modélisant le distributeur

FSM, les LTS ne distinguent ni les entrées ni les sorties. De manière simplifiée, un LTS est composé d'un ensemble d'états (dont un état initial), entre lesquels sont définies des transitions étiquetées (via une relation de transition) dont une transition particulière τ pour modéliser les transitions internes et sans synchronisation. La spécification du distributeur de boissons ci-dessus est modélisée par le système de transitions étiquetées de la figure 2.6.

2.5 Couverture du graphe de flot de contrôle

Les critères de couverture d'un graphe de flot de contrôle sont exprimés entièrement en termes de traces d'exécution du programme sous test à travers son graphe de flot de contrôle. Un critère de couverture peut être calculé à partir d'informations structurelles liées au flot de contrôle ou à partir d'informations liées à l'analyse de flot de données. L'exécution d'un programme avec une donnée de test correspond à un parcours du graphe de flot de contrôle. Chaque exécution correspond, donc, à un chemin du graphe allant du noeud d'entrée vers le noeud de sortie. Un tel chemin est dit chemin d'exécution ou chemin de calcul. Le nombre de chemins d'exécution d'un programme peut être infini, à cause de la présence de boucles infinies dans le

graphe. On appelle exécution de test tout chemin d'exécution avec une donnée de test.

Dans un graphe de flot de contrôle, certains éléments ne font jamais partie d'un chemin d'exécution. Ces éléments (une instruction, une branche, un chemin) sont appelés éléments infaisables. Ainsi, une instruction infaisable est une instruction pour laquelle il n'y a aucun jeu de tests qui permet de l'exécuter. Un bloc d'instructions infaisables est appelé communément *code mort* . Par ailleurs, une branche infaisable est une branche dont le parcours n'est jamais déclenché par une quelconque donnée de test. Cette situation vient du fait que la condition de transfert de contrôle via cette branche n'est jamais satisfaite au moment de l'évaluation de l'expression booléenne dans le noeud.

Exemple 2.5.1 Dans le graphe de la figure 2.2, le chemin (a, b, d, f, g) est un chemin d'exécution qui correspond à l'exécution du programme pour $x = -5$ et $y = -5$. De même, le chemin (a, b, d, g) est un chemin d'exécution qui correspond à une exécution possible du programme pour les entrées $x = -5$ et $y = -6$. Par ailleurs, ce graphe ne contient pas de chemins infaisables.

Les chemins infinis et les éléments infaisables dans un graphe de flot de contrôle empêchent la satisfaction de certains critères (atteindre la couverture complète). Ils sont à l'origine de la notion d'applicabilité finie d'un critère introduite dans [63]. L'applicabilité finie est définie comme étant l'exigence pour qu'un critère soit toujours satisfait par un jeu de tests. Un critère satisfaisant cette exigence est dit applicable de manière finie. Dans la suite de cette section, nous passons en revue l'ensemble des critères les plus connus et les plus utilisés pour la couverture des graphes de flot de contrôle. Nous présentons deux familles de critères de couverture du graphe de flot de contrôle : les critères basés sur l'analyse de flot de contrôle et les critères basés sur l'analyse de flot de données

2.5.1 Critères basés sur l'analyse de flux de contrôle

Sous l'hypothèse que l'exécution de tous les chemins possibles d'un programme permet de révéler toutes les fautes dans celui-ci, le critère de couverture de tous les chemins constitue le critère le plus fort (grâce à son caractère exhaustif). Dans des programmes contenant des boucles infinies, la satisfaction de ce critère est inaccessible.

Des critères plus faibles et plus accessibles comme le critère de couverture des instructions et le critère de couverture des branches sont largement utilisées en milieu industriel. Ces critères mesurent respectivement le pourcentage d'instructions ou de branches du programme qui ont été exécutées au moins une fois durant le test. D'autres critères intermédiaires, tels que la couverture des chemins de longueur k et la couverture des LCSAJ (Linear Code Séquence and Jump) [62], ont comme principale utilité d'offrir des paliers mesurables entre la couverture des branches et celle des chemins.

Couverture des chemins

Définition 2.5.1 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère de couverture des chemins si et seulement si $\mathcal{P}$ contient tous les chemins d'exécution allant du noeud d'entrée au noeud de sortie du graphe de flot de contrôle.*

La satisfaction de ce critère est, souvent impossible à cause de la présence de boucles qui nécessite de couvrir une infinité de chemins. En effet, un test doit avoir un temps d'exécution fini. Ce qui suppose que le jeu de tests correspondant est fini. Lorsque le graphe contient un nombre infini de chemins distincts, la satisfaction du critère nécessite d'exécuter une infinité de cas de test. Ce qui est impossible en pratique. Cet inconvénient est contourné de deux manières : La première solution consiste à limiter le nombre de boucles à parcourir afin de définir un critère satisfiable. La question est comment peut-on identifier le nombre de boucles minimal. La seconde consiste à définir d'autres critères moins forts que le critère de couverture de tous les chemins mais plus réalistes, en couvrant des éléments finis du graphe

(instructions, décisions,...)

Couverture des instructions (SC)

Une exigence de base de ce critère est que toutes les instructions du programme soient couvertes par des exécutions sur des données de test. Une couverture complète des instructions n'est pas toujours possible à cause de l'existence éventuelle d'instructions qui ne s'exécutent jamais, *le code mort*. Puisque les instructions correspondent à des arcs dans le graphe de flot de contrôle, la couverture des instructions peut être définie comme suit :

Définition 2.5.2 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère de couverture des instructions si et seulement si pour tout arc e du graphe de flot de contrôle, il existe au moins un chemin d'exécution $p \in \mathcal{P}$ tel que $e \in p$.*

Exemple 2.5.2 La couverture de toutes les instructions du programme (**Fig.** 2.1) est assurée par un seul chemin d'exécution (a, b, d, f, g) qui parcourt toutes les instructions. Elle est satisfaite par l'exécution de $x = -5$ et $y = -5$.

Dans l'industrie, il existe une variante de ce critère connue sous le nom de couverture de lignes. Elle consiste à couvrir la totalité (ou un certain pourcentage) des lignes de code dans un programme. Ce critère prend un sens pour des langages comme l'assembleur par exemple où une instruction est synonyme de ligne de code. Pour les langages évolués, en revanche, il est plus sensé de couvrir toutes les instructions que de couvrir toutes les lignes du code, car un programme C, par exemple, peut être formaté sur une seule ligne.

Bien que l'exécution de chaque instruction soit nécessaire durant le test, le critère de couverture des instructions est considéré comme un critère faible. En effet, un jeu de tests qui couvre toutes les instructions n'assure pas, nécessairement, la couverture de certains transferts de contrôle dans le programme. Par exemple, lorsque la partie ELSE d'une instruction IF-THE-ELSE est vide, l'exécution de la partie THEN suffit à couvrir l'instruction alors que la branche ELSE n'est pas examinée. D'où, le critère

de couverture des branches qui, en plus de la couverture de toutes les instructions, nécessite la couverture de tous les transferts de contrôle. En voici la définition.

Couverture des branches

Définition 2.5.3 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère de couverture des branches si et seulement si $\mathcal{P}$ satisfait la couverture des instructions et que pour tous les noeuds du graphe $n \in N$, il existe au moins un chemin d'exécution $p \in \mathcal{P}$ qui passe par chacune des branches du noeud n .*

Exemple 2.5.3 La couverture de toutes les branches (100%) du programme (**Fig.** 2.1) est assurée par deux chemins d'exécution (a, b, d, f, g) et (a, c, e, f, g) qui passent par tous les noeuds du graphe et par toutes les branches issues de ces noeuds. Deux entrées de test sont nécessaires pour assurer cette couverture $(-5, -5)$ et $(5, 5)$.

Il est évident que le critère de couverture des branches inclut le critère de couverture des instructions. En effet, le critère de couverture de branches est plus fort que le critère de couverture des instructions car si tous les arcs du graphe sont couverts, tous les noeuds seront nécessairement couverts. Par conséquent, tout jeu de tests qui satisfait le critère de couverture de toutes les branches doit obligatoirement satisfaire le critère de couverture des instructions.

Le critère de couverture des instructions (respectivement branches) ne sont pas toujours applicables de manière finie, notamment à cause de l'existence d'instructions (respectivement branches) infaisables. Des versions applicables de façon finie peuvent être définies en considérant exclusivement les éléments faisables. Cependant, ce problème est indécidable [60].

Couverture LCSAJ

Cette mesure de la couverture est une variante de la couverture des chemins. A l'origine, ce critère est défini directement sur le code du programme. Son principe est d'assurer la couverture des portions de code appelés *Linear Code Sequence And Jump* (LCSAJ). Une LCSAJ est toute portion contiguë du code qui s'exécute de

façon séquentielle et qui se termine par un saut. Plusieurs LCSAJs peuvent être construits par concaténation de plusieurs LCSAJs consécutifs pour donner lieu à des critères de couverture de plus en plus forts (2 LCSAJs, 3 LCSAJs,....., n LCSAJs). Cette progression offre des moyens de mesure de la couverture réalisables et toujours utilisables en se rapprochant de la couverture des chemins.

Les métriques de couverture définies par *Woodward et al* dans [62], appelées *Test Effectiveness Ratio* (TER_i), permettent d'établir une hiérarchie de critères dont le plus faible est le critère de couverture des instructions (TER_0). TER_1 correspond au critère de couverture des branches, TER_2 mesure la couverture d'une LCSAJ, TER_3 mesure la couverture de 2 LSCAJs. De manière générale, TER_{n+1} mesure la couverture de n LCSAJs.

2.5.2 Critères basés sur l'analyse de flot de données

D'autres critères ont été proposés dans la littérature [11,51] à base de l'analyse du flot de données. Dans ces critères, la couverture est définie en s'appuyant sur l'analyse des sous chemins reliant les définitions des variables à leurs utilisations potentielles. Cette analyse met en évidence la manière dont les valeurs sont associées aux variables et comment ces associations peuvent affecter l'exécution du programme, et se focalise sur les occurrences des variables dans le programme. Toute occurrence d'une variable est considérée soit comme une *définition* (la variable est liée à une valeur) soit comme une *utilisation* (la variable est référencée).

L'utilisation d'une variable figure soit dans un prédicat dont l'évaluation détermine le chemin d'exécution (*utilisation-prédicat*), soit dans un calcul pour le calcul d'une autre variable ou comme valeur de sortie (*utilisation-calcul*).

L'occurrence d'une variable x comme définition dans un noeud u atteint l'occurrence de la même variable en tant qu'utilisation dans le noeud v si et seulement s'il existe un chemin d'exécution $p = (u, w_1, w_2, ...w_n, v)$ tel que le chemin $p = (w_1, w_2, ...w_n)$ ne comprend aucune occurrence de x comme définition et l'occurrence de x dans v est globale. De manière similaire, l'occurrence d'une variable x comme définition dans un noeud u atteint l'occurrence de la même variable en

tant qu'utilisation-prédicat dans le noeud v si et seulement s'il existe un chemin $p = (u, w_1, w_2, ...w_n, v)$ de u à v tel que le chemin $p = (w_1, w_2, ...w_n)$ ne comprend aucune occurrence de x comme définition et s'il y a une *occurrence-prédicat* de x dans v.

Plusieurs variantes de ces critères ont été proposées [17, 51].

Couverture de toutes les définitions (all-defs)

Définition 2.5.4 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère toutes les définitions si et seulement si pour toutes les occurrences-définitions d'une variable x telle qu'il existe une utilisation de x atteignable à partir de la définition, il existe au moins un chemin $p \in \mathcal{P}$ tel que p contient un sous chemin dans lequel la définition de x atteint d'autres occurrences-utilisation de x.*

La satisfaction de ce critère nécessite que le jeu de tests couvre toutes les occurrences-définitions. Autrement dit, les cas de test doivent couvrir un chemin dans lequel une définition atteint l'utilisation de cette définition.

Couverture de toutes les utilisations (all-uses)

Définition 2.5.5 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère toutes les utilisations si et seulement si pour toutes les occurrences-définitions et toutes les occurrences-utilisations dune variable x telle qu'il existe une utilisation de x atteignable à partir de la définition, il existe au moins un chemin $p \in \mathcal{P}$ tel que p contient un sous chemin dans lequel la définition de x atteint l'utilisation de x.*

La distinction entre utilisation-calcul et utilisation-prédicat pour les occurrences des variables a donné lieu à d'autres critères [51].

Couverture de tous les chemins définitions-utilisations (all-du-paths)

Définition 2.5.6 *Un ensemble $\mathcal{P}$ de chemins d'exécution satisfait le critère tous les chemins définitions-utilisations si et seulement si pour toutes les occurrences-définitions d'une variable x et pour tous les chemins q à travers lesquels une uti-*

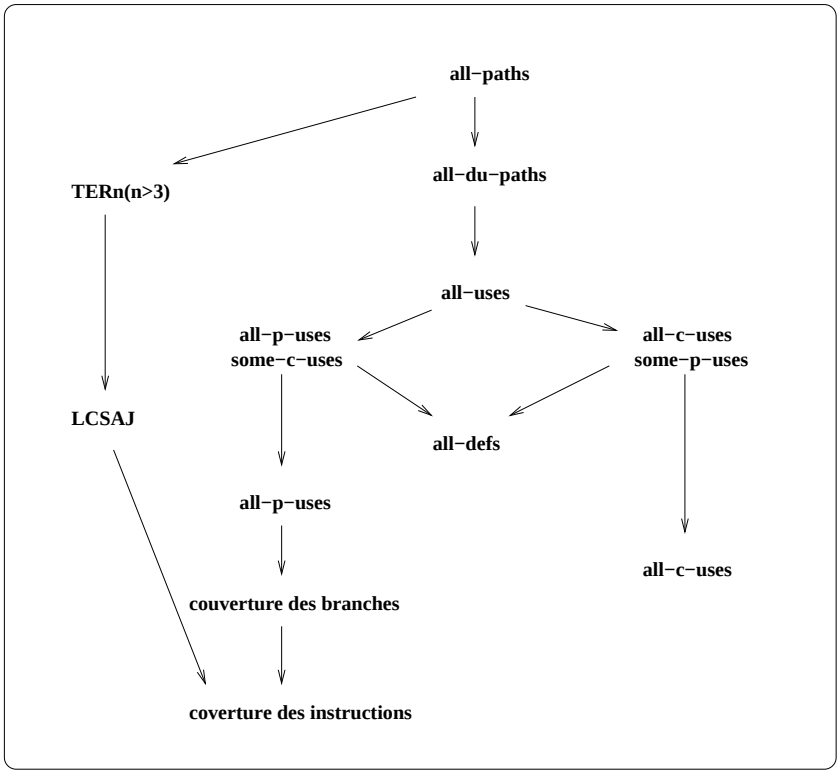

FIG. 2.7 – Inclusion des critères

lisation de x est atteignable, il existe au moins un chemin $p \in \mathcal{P}$ tel que q est un sous-chemin de p, et q ne contient pas de cycles.

D'autres critères basés sur d'autres types de dépendances entre les définitions et les utilisations des variables ont été proposés (cf. [31, 41]).

Les différentes relations d'inclusion entre les critères basés sur l'analyse du flot de contrôle et les critères basés sur l'analyse de flot de données sont résumées dans la figure 2.7.

2.6 Couverture des expressions booléennes

Dans [58, 59], le terme couverture de toutes les décisions (DC) est employé pour désigner le critère de couverture de toutes les branches. En effet, une branche dans un graphe de flot de contrôle correspond à une des deux évaluations possibles de la décision que contient un noeud du graphe. Basés sur une décomposition d'une décision (branche) en conditions atomiques, appelées *conditions* [58] ou *clauses* [43],

d'autres critères affinant le critère de couverture des décisions ont été proposés.

Couverture des Conditions (CC)

Définition 2.6.1 *Un ensemble de chemins d'exécution $\mathcal{P}$ satisfait le critère de couverture des conditions si et seulement si $\mathcal{P}$ satisfait la couverture de toutes les instructions et que pour chaque décision, chacune des conditions ait pris toutes les valeurs possibles.*

Les critères de couverture des conditions et de couverture des décisions (branches) ne sont pas comparables. Autrement dit, un jeu de tests satisfaisant l'un des critères ne signifie guère qu'il satisfait l'autre et vice et versa. La combinaison des deux donne lieu à la définition d'un troisième critère qui satisfait les deux à la fois. Ce critère s'appelle Couverture des Conditions/Décisions (DCC)

Couverture de Conditions/Décisions (DC/C)

Définition 2.6.2 *Un ensemble de chemins d'exécution $\mathcal{P}$ satisfait le critère de Couverture des Conditions/Décisions si et seulement si $\mathcal{P}$ satisfait la couverture de toutes les instructions, et pour chaque décision, chacune des conditions prend toutes les valeurs possibles et chaque décision prend toutes les valeurs possibles.*

Couverture de conditions multiples (MCC)

Définition 2.6.3 *Un ensemble de chemins d'exécution $\mathcal{P}$ satisfait le critère de Couverture des Conditions multiples si et seulement si $\mathcal{P}$ satisfait la couverture de toutes les instructions, et que pour chaque décision, toutes les combinaisons possibles de ses conditions aient été testés au moins une fois.*

Le critère MCC est le critère le plus fort. Cependant, une explosion dans le nombre de cas de test est vite atteinte. Une décision ayant n conditions nécessite 2^n cas de test ce qui nécessite un coût de calcul énorme. De plus, l'exhaustivité des combinaisons est inutile dans certaines conditions qui contiennent des dépendances entre les variables. Une solution qui minimise le nombre de combinaisons consiste

donc d'analyser les dépendances entre les variables dans les conditions constituant la décision, afin de retenir que les combinaisons dans lesquels les conditions influent sur la valeur de la décision. Ce critère s'appelle MC/DC

Couverture de condition / Décision modifiée (MC/DC)

Définition 2.6.4 *Un ensemble de chemins d'exécution $\mathcal{P}$ satisfait le critère de couverture MC/DC si et seulement si $\mathcal{P}$ satisfait la couverture de toutes les instructions, et que pour chaque décision, toutes les combinaisons possibles des conditions dont la variation influe de façon indépendante sur la valeur de la décision aient été testés au moins une fois.*

Une condition influe de façon indépendante sur la valeur d'une décision si le seul fait de faire varier la condition en fixant les autres change la valeur de la décision. Ce critère est similaire au critère FPC [58] utilisé pour la couverture des spécifications booléennes. La seule différence entre les deux critères est que ce dernier n'exige pas de fixer les autres conditions d'une décision, ce qui rend le masquage d'une condition par les autres possible. Le critère MC/DC est l'un des critères les plus utilisés dans les applications industrielles, notamment les applications critiques dans des domaines comme l'aéronautique. Ce critère est à la base de la norme DO178-B. Les relations entre les critères ci-dessus sont résumées dans la hiérarchie d'inclusion [58] de la figure 2.8.

Norme DO-178B

La norme DO-178B "Software Considérations in Airborne Systems and Equipment Certification" (Réglementation pour le développement de logiciels dans le secteur aéronautique) est le standard pour le développement de logiciels dans le domaine aéronautique. Les prescriptions de la RTCA[1] dans le FAA Advisory Circular AC20-115B sont à la base de la certification des logiciels par le biais des procédés de développement des logiciels.

[1]Radio Technical Commission for Aeronautics, Inc.

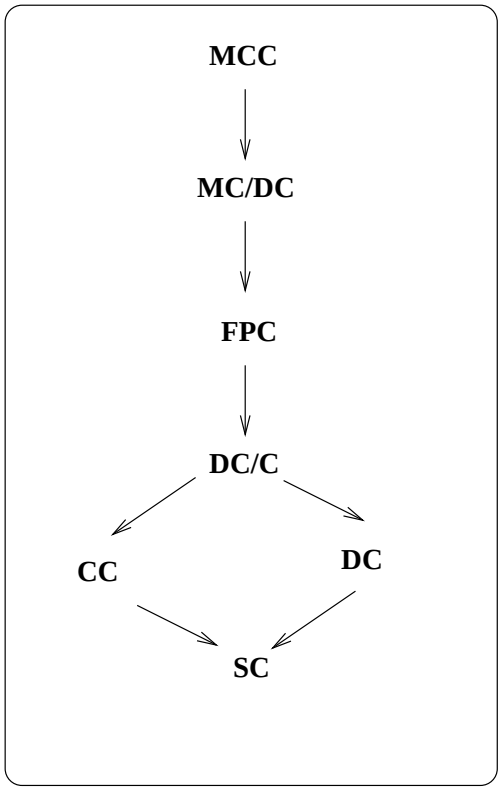

FIG. 2.8 – Hiérarchie des critères de couverture de flot de contrôle

DO-178B est devenu une norme de facto produite par la RTCA, le FAA's Advisory Circular AC20-115B a établi le DO-178B qui est le moyen admis pour la certification de tous les nouveaux logiciels d'aviation. RTCA DO-178B[2] définit les directives pour le développement de logiciels d'aviation aux États-Unis. Le succès du DO-178B s'est étendu aux autres secteurs et il est la base de nombreux processus de certification utilisés aujourd'hui. Pour la certification DO-178B, certains documents et justificatifs doivent être fournis en accompagnement du logiciel concerné (entre autres le protocole de test).

Les différents niveaux de DO-178B (DO-178B-Levels) correspondent aux conséquences des éventuelles erreurs de logiciels :

- Catastrophiques (niveau A),
- Dangereuses / difficiles (niveau B),
- Majeures (niveau C),

[2]L'EUROCAE ED-12B est l'équivalent européen du DO-178B.

- Mineures (niveau D)
- Sans effet sur la sécurité (niveau E).

Selon le niveau, les couvertures suivantes doivent être effectuées à 100% :

- **Niveau A :**
  - Couverture de condition / Décision modifiée (Modified Condition Decision Coverage : MC/DC)
  - Couverture de condition / Couverture de décision (DC/C)
  - Couverture des instructions (SC)
- **Niveau B :**
  - Couverture de condition / Couverture de décision (DC/C)
  - Couverture des instructions (SC)
- **Niveau C :**
  - Couverture des instructions (SC)

2.7 Couverture des fautes

Le bien-fondé des critères de couverture des graphes de flot de contrôle ne repose que sur l'intuition que plus un test met en en jeu d'éléments, et de combinaisons d'éléments du programme, plus il a de chance de découvrir des fautes. Dans le but de mieux cerner cette intuition, d'autres critères reposent sur l'aptitude d'un jeu de tests à détecter des fautes, à condition de pouvoir les caractériser via des modèles.

Test par mutation

Le test par mutation [9, 12] est une méthode de test basée sur la couverture des fautes les plus probables, caractérisées par un modèle de fautes. L'objectif du test par mutation est d'évaluer l'aptitude d'un jeu de tests à détecter les fautes dans un programme. Cette technique est basée sur l'hypothèse suivante :

Si un jeu de tests est incapable de distinguer les comportements de deux programmes p et m tels que p diffère de m par une faute usuelle, alors le jeu de tests est dit insuffisant pour déterminer si le programme contient une faute.

Pour cela, le programme p et son mutant m sont exécutés sur un jeu de tests et leurs comportements sont comparés via un oracle. Le mutant est dit tué si son comportement diffère de celui du programme d'origine. Sinon, le mutant est dit équivalent au programme d'origine.

L'aptitude d'un jeu de tests à révéler les fautes est évaluée par rapport au nombre de mutants tués. Ce dernier est défini par un taux, calculé comme le rapport entre le nombre de mutants non-équivalents tués au nombre de mutants créés (un nombre inférieur ou égal à 1). Il mesure à quel point le programme est sensible au changement de code. Un jeu de tests est dit adéquat vis-à-vis du critère de couverture des fautes si le score mutationnel est égal à 1.

Les principaux avantages de la mutation sont la facilité de la mise en oeuvre et l'automatisation du processus de test (phase de sélection et de soumission, phase de génération des mutants, phase de comparaison du programme et des mutants). Cependant, il y a deux principales limitations :

1. Les mutants peuvent être fonctionnellement équivalents par rapport au programme original auquel cas aucun jeu de tests ne peut les distinguer. Par exemple, si une variable n'est jamais nulle dans une décision dans laquelle elle est comparée à "0" en utilisant l'opérateur ">", le mutant dans lequel cet opérateur est remplacé par ">=" sera équivalent. En plus, Il n'existe aucun moyen pour séparer l'effet de la non-minutie d'un jeu de tests de celui lié au mutants équivalents.
2. Son coût exorbitant, à cause du nombre potentiel de mutants pour des gros programmes. Néanmoins, plusieurs variantes du test par mutation existent dans la pratique et qui génèrent moins de mutants que la mutation standard.
   (a) *la mutation contrainte*, se concentre sur la génération d'un sous-ensemble de mutants basés sur les opérateurs.
   (b) *la mutation sélective* utilise un sous-ensemble de mutants sélectionnés de manière aléatoire.
   (c) *la mutation faible* consiste en l'utilisation de l'état du programme pendant

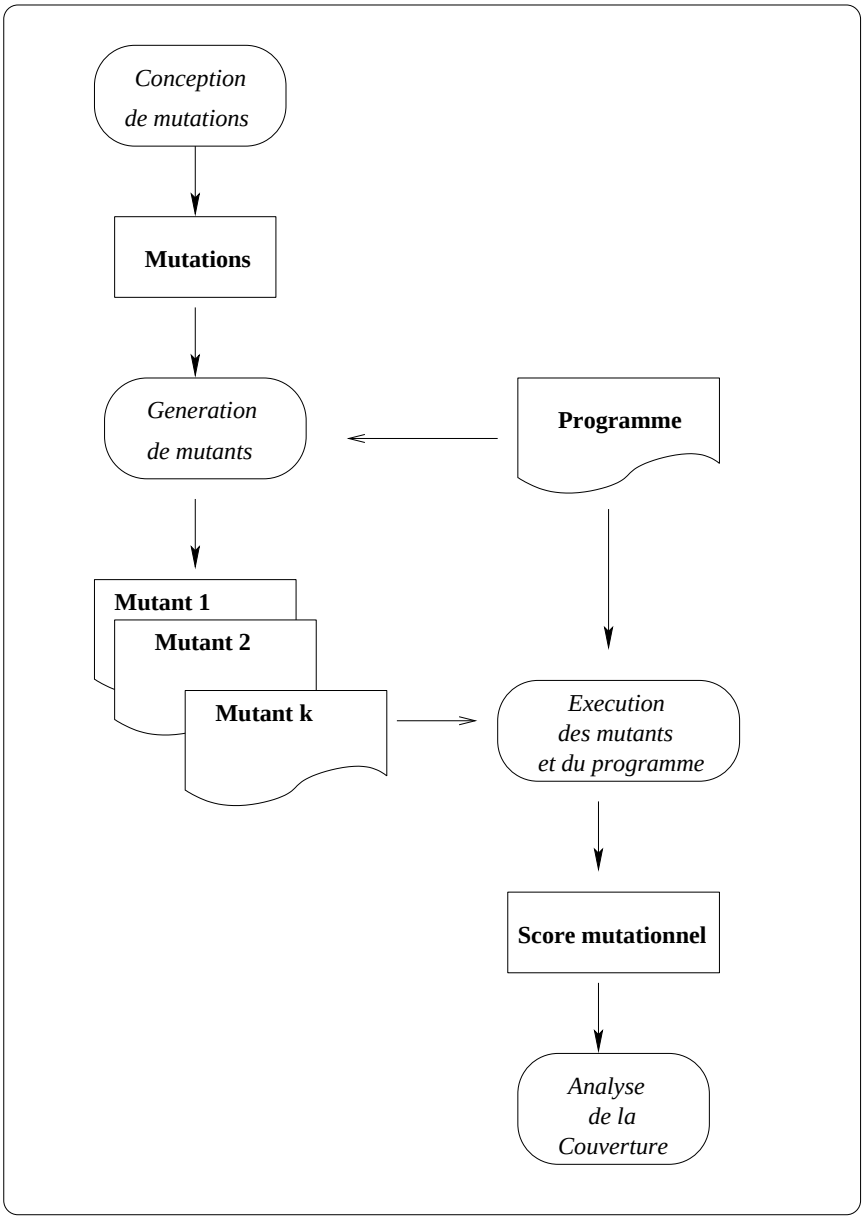

FIG. 2.9 – Processus de mutation

l'exécution plutôt que sa sortie pour distinguer les mutants.

Test à base des graphes causes-effets

Les graphes causes-effets sont utilisées à la fois pour modéliser les expressions booléennes dans les spécifications ou dans les programmes. De nombreux critères de couverture visant la détection de certaines classes de fautes récurrentes dans les expressions booléennes ont été proposés par *Tai et al* [53] sur les graphes cause-effets.

Deux des principaux critères définis sur ls graphes causes-effets sont le critère BOR et le critère BRO. Ces critères sont utilisés principalement comme des critères de sélection pour la génération des jeux de tests qui visent à mettre en évidence des fautes dans les expressions booléennes. Nous donnerons ci-dessous les définitions informelles de ces deux critères. Des définitions formelles avec des études de cas peuvent être trouvées dans [53].

Le critère **BOR** (*Boolean OperatoR*) est un critère qui vise à détecter des fautes liées aux opérateurs booléens au niveau des expressions booléennes (AND, OR et NOT) comme l'inversion des opérateurs AND et OR, ou l'omission/l'ajout d'un opérateur NOT. Afin d'illustrer ce critère, considérons l'expression booléenne (voir l'expression 2.1) dont le graphe causes-effets est donné dans la figure 2.10.

$$((E1 < E2) \wedge (E3 \geq E4)) \vee (E5 = E6) \tag{2.1}$$

Le jeu de tests $TS1 = \{(t,t,f),(t,f,t),(t,f,f),(f,t,f)\}$ assure une couverture BOR de cette expression. En effet, tout changement ou remplacement d'un opérateur est détecté par ce jeu de tests.

Le critère **BRO** (*Boolean and Relational Operator*) est un critère plus fort que le critère précédent. Outre la détection des fautes relatives aux opérateurs booléen, ce critère vise la détection des fautes au niveau des opérateurs relationnels ($\leq, \geq, =, \neq, <, >$). Ainsi, le jeu de tests $TS1$ est insuffisant pour révéler les éventuelles fautes commises au niveau des opérateurs relationnels. En revanche, le jeu de tests $TS2 = \{(>,>,<),(>,=,<),(<,<,=),(<,<,<),(=,=,>),(>,=,>)\}$ assure une couverture BRO de l'expression 2.1. Dans [53], un algorithme permet de

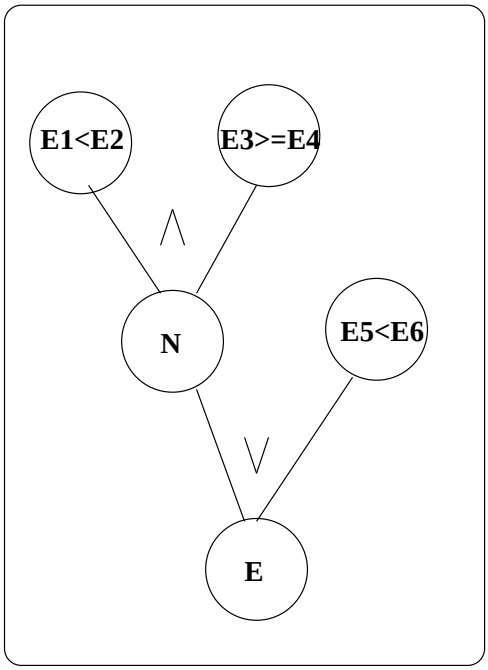

FIG. 2.10 – Graphe causes-effets

construire, à partir des graphes causes-effets des contraintes minimales pour assurer cette couverture.

2.8 Conclusion

En conclusion, les problèmes de couverture sont des problèmes d'une grande importance dans le test logiciel. La plupart des travaux de critères de couverture présentés dans ce chapitre sont bien adaptés aux langages impératifs (couverture du graphe de flot de contrôle, couverture des expressions booléennes). Certains critères, exprimées sur des formalismes comme les graphes causes-effets, sont applicables au niveau spécification.

Il existe une littérature très riche concernant la couverture des machines d'états finis ou des systèmes de transitions étiquetés, dont les critères nous semblent loin du cadre que nous traitons. Les critères basés sur les modèles de fautes offrent un moyen de couverture indépendant du paradigme dans lequel le programme est exprimé.

Dans le chapitre suivant, nous passons en revue l'état de l'art des logiciels synchrones notamment en termes de validation par le test.

Chapitre 3

Test des programmes écrits en LUSTRE

3.1 Caractéristiques des logiciels réactifs

Un système informatique classique, dit transformationnel, dispose de toutes ses données d'entrée à l'initialisation. Il effectue les traitements nécessaires sur ces données et fournit, à sa terminaison, des résultats. Dans un logiciel interagissant avec l'environnement, l'interaction est un processus cyclique tel que chaque cycle correspond à un échange actions-réactions. Selon le mode de contrôle de la cadence de l'interaction, on distingue deux classes [4] : les logiciels *conversationnels* et les logiciels *réactifs (ou réflexes).*

Dans les logiciels *conversationnels*, l'environnement émet des requêtes vers le logiciel. Celui-ci choisit s'il doit ou non y répondre, ainsi que le moment où il doit fournir la réponse. Il est, donc, maître de l'interaction, et l'environnement attend d'être servi. Des exemples de ce type de systèmes incluent les systèmes d'exploitation, les bases de données, les systèmes réseau.

Dans les logiciels *réactifs*, c'est l'environnement qui est maître de l'interaction : le rôle du système est de réagir de façon continue aux stimuli externes en produisant des réponses instantanées. On trouve ces logiciels dans les domaines où l'environnement ne peut pas attendre (systèmes temps réel), comme le contrôle de processus

industriels, les systèmes embarqués et les pilotes de dispositifs. Le déterminisme est une propriété souvent désirable pour ces types de logiciels : les résultats émis doivent dépendre uniquement des informations reçues et de leur enchaînement temporel.

En résumé, les logiciels réactifs sont des programmes qui réagissent à l'environnement à la vitesse de celui-ci alors que les autres logiciels réagissent à leur propre vitesse. Les logiciels conversationnels sont asynchrones et souvent non-déterministes, et les logiciels réactifs sont parfaitement synchrones et souvent déterministes [6].

3.2 Logiciels réactifs synchrones

3.2.1 Principes

Un système réactif n'est pas uniquement descriptible à l'aide des relations transformationnelles, spécifiant des sorties à partir des entrées, mais plutôt par des liens entre sorties et entrées via leurs combinaisons possibles dans le temps. Dès lors, c'est la combinaison de descriptions englobant des séquences complexes d'événements, actions, conditions et flots de données qui permet de synthétiser le comportement d'un système réactif.

La complexité des systèmes réactifs découle essentiellement de la nature des réactions conséquentes à des occurrences d'événements discrets. La modélisation des systèmes réactifs n'est donc pas une activité facile. Plusieurs formalismes ont été proposés pour la réaliser. Nous retenons les *statecharts* [22], ESTEREL [7], LUSTRE [19], qui relèvent de l'approche synchrone où les sorties sont considérées comme instantanées, synchrones avec les entrées.

Ces langages réactifs sont basés sur le modèle du synchronisme parfait, dans lequel les processus sont capables, au niveau conceptuel, de s'exécuter et d'échanger des données en un temps nul. Le modèle sous-jacent est équivalent au modèle "*zéro-délai*" des circuits électroniques. Bien que ce paradigme soit peu naturel pour des programmeurs classiques, c'est un standard pour les théoriciens du contrôle, les concepteurs de circuits électroniques, ou les utilisateurs de contrôleurs programmables.

Dans le paradigme réactif, un événement est un signal. Du point de vue logiciel, un programme réactif s'exécute en une séquence ininterrompue d'itérations appelées tics (ou cycles), qui sont des réactions aux signaux d'entrée. Dans la pratique, un fonctionnement synchrone où le temps de calcul de sorties est nul n'existe pas. En effet, quelle que soit la puissance de calcul utilisée pour l'exécution du logiciel, le calcul des sorties requiert toujours une certaine durée.

Selon l'hypothèse dite de *synchronisme*, un logiciel réactif est considéré comme étant synchrone si et seulement si le temps de réponse du logiciel est inférieur au temps minimum nécessaire à l'observation de toutes les évolutions significatives de l'environnement. Ce temps dépend de la nature des systèmes.

Dans la pratique, l'hypothèse du synchronisme est vérifiée si les réactions sont assez rapides pour qu'aucun signal provenant de l'environnement ne soit perdu. Autrement dit, le programme réagit à ses entrées en calculant ses sorties dans un temps borné par la durée de cycle d'une horloge globale. Un cycle définit une séquence d'instants qu'on appelle horloge de base.

3.2.2 Fonctionnement du modèle synchrone

Dans le modèle synchrone, le comportement d'un logiciel peut être caractérisé par une boucle infinie au cours de laquelle, les trois étapes suivantes s'enchaînent de manière séquentielle :

1. le logiciel reçoit les entrées de son environnement ;
2. le logiciel calcule les sorties ;
3. le logiciel émet les sorties calculées vers son environnement.

Ce mode de fonctionnement permet de dater précisément chaque événement interne du programme par rapport au flot des événements externes. Ceci permet, en théorie, de définir une suite d'instants discrets où, à chaque instant t_i, le logiciel reçoit une entrée e_i de l'environnement puis émet une sortie s_i, avec un temps de calcul des sorties supposé nul. Ce comportement est illustré dans la figure 3.1

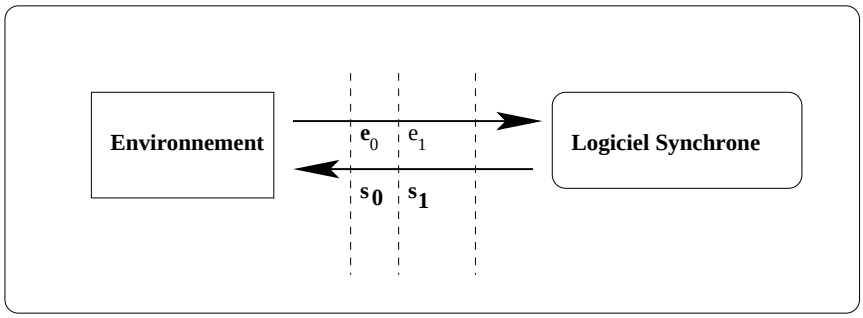

FIG. 3.1 – Fonctionnement d'un logiciel synchrone

Il existe deux styles de programmation des logiciels synchrones : le style impératif basé sur une expression explicite du flot de contrôle et le style déclaratif basé sur une expression en termes de transformation de flots de données.

La suite de ce chapitre sera consacrée aux langages du deuxième style.

3.2.3 Flots de de données synchrones

Le paradigme "flot de données" est à la fois un concept de programmation et une technique d'implantation. En spécification comme en programmation, le flot de données laisse entendre l'échange d'information (les données) entre des entités (fonctions) qui manipulent des données et effectuent des calculs sur elles. Par conséquent, dans un programme basé sur le flot de données, une étape est engagée lorsque d'autres étapes lui fournissent ses entrées directement à partir de leurs sorties. Les modules du programme sont alors toujours prêts à effectuer leurs opérations. Une fois que toutes les données nécessaires arrivent au module, l'opération est lancée automatiquement.

Le paradigme "*flot de données*" met l'accent sur la détermination des entrées en exigeant que les sorties d'une étape soient explicitement liées aux entrées d'une (ou des) autre(s) étape(s). Il ne tient pas compte de l'ordonnancement des étapes, parce que le moment où toutes les entrées d'une étape sont disponibles est déterminé par les temps que les diverses étapes mettent pour délivrer leurs sorties. Par conséquent, certaines étapes peuvent obtenir toutes leurs entrées plus tôt que d'autres dans un ordre moins restrictif que le flot de contrôle. Ces caractéristiques rendent ce para-

digme plus convenable aux applications dans lesquelles la détermination appropriée des entrées est la plus importante, comme dans les applications temps réel.

Les systèmes réactifs synchrones sont, particulièrement, utilisés dans des domaines manipulant des flots de données tels que les systèmes de contrôle, les circuits électroniques,.... Traditionnellement, les concepteurs dans ces domaines modélisent leurs systèmes en utilisant des schémas à base d'opérateurs (switchs, circuits analogiques,...). Ces schémas servent à transformer des flots de données. Au plus haut niveau, ils utilisent des équations booléennes et des fonctions de transfert avec des structures "diagrammes de blocs" et des équations différentielles pour capturer le comportement de ces réseaux.

3.3 Le langage LUSTRE

LUSTRE [10] est un langage déclaratif à flots de données synchrones munis d'opérateurs temporels permettant de se référer à des valeurs passées des données. LUSTRE a été développé pour la spécification et la programmation des logiciels réactifs synchrones dont le comportement est cyclique. Il est utilisé dans des domaines variés du secteur industriel (aéronautique, énergie, transport,...) car il possède une syntaxe graphique proche de celle utilisée dans ces domaines pour décrire les systèmes matériels (diagrammes de blocs, réseaux d'opérateurs...).

Les données sont des flots de valeurs. La dimension temporelle, représentée par une horloge dite horloge de base, met en relation le temps et le rythme des données dans le flot. Un *flot* est une paire constituée d'une séquence infinie de valeurs d'un type donné et d'une horloge représentant une séquence de temps discrets. Au n-ème instant de l'horloge, le flot prend la n-ème valeur. Le langage reconnaît les types de base *booléen* (**bool**), *entier* (**int**), *flottant* (**real**) ainsi que le type *énumération* (**tuple**).

Une constante C est définie comme un flot de valeurs $(c, c, ..., c, ...)$ associé à une horloge de base. De même, une variable X est définie comme un flot de valeurs $(x_1, x_2, ..., x_n, ...)$ associé à une horloge de base. Le langage LUSTRE supporte deux

types d'opérations :

1. Les opérations combinatoires dont les flots d'entrée et de sortie sont définis au même instant de l'horloge
2. Les opérations temporelles dont les flots d'entrée et de sortie sont définis à des instants différents de l'horloge (mémoire).

3.3.1 Opérateurs arithmétiques et logiques

Les opérations usuelles sur les types de base du langage manipulent des flots définis sur la même horloge. Elles utilisent des opérateurs qui se divisent en quatre classes :

- Les opérateurs **logiques** *and*, *or* et *not* qui manipulent des flots booléens.
- Les opérateurs **arithmétiques** $+, *, /, div, mod$, et $"-"unaire$ et $"-"binaire$ combinent des flots numériques (entiers et/ou flottants)
- Les opérateurs **relationnels** $>, >, =>, <=, =, <>$ pour comparer des flots numériques.
- L'opérateur **conditionnel** $if - then - else$ pour choisir entre deux flots booléens ou numériques.

Selon le nombre de flots en entrée (opérandes), on distingue les opérateurs unaires, binaires et ternaires :

- Unaires (not et $"-"unaire$),
- Binaires (and, or,$+, -, *, /, div, mod$, $>, >, =>, <=, =, <>$)
- Ternaires ($if - then - else$).

Les opérateurs binaires et l'opérateur ternaire s'appliquent point à point (*pointwise*) sur les flots en entrées. Ainsi, si X et Y décrivent respectivement les deux flots $(x_1, x_2, ..., x_n, ...)$ et $(y_1, y_2, ..., y_n, ...)$ alors l'expression $if \;\; X \geq 0 \;\; then \;\; Y + 1 \; else \; 0$ décrit le flot dont la n-ème valeur est donnée par $if \; x_n \geq 0 \; then \; y_n{+}1 \; else \; 0$.

3.3.2 Opérateurs temporels

Le langage LUSTRE propose également des opérateurs, appelés opérateurs temporels, dont les flots d'entrée et le flot de sortie sont associés à des instants distincts de l'horloge de base. La différence entre les deux instants introduit ce qu'on appelle un *retard*.

L'opérateur"*précédent*", noté "*pre*", agit comme une mémoire. Si E est une expression dénotant la séquence de valeurs $(e_1, e_2, ..., e_n, ...)$ alors $pre(E)$ dénote la séquence des valeurs $(nil, e_1, e_2, ..., e_n, ...)$ où nil est une valeur indéterminée. En d'autres termes, l'opérateur pre permet d'accéder, à un instant t, à la valeur prise par une expression à l'instant $(t-1)$. De manière générale, la notation $pre^{n}(E)$ décrit l'expression $pre(pre(.....pre(E)))$, composée de n pre imbriqués, et permet d'accéder à la valeur de E à l'instant $(t-n)$.

	t_0	t_1	t_2	.....	t_n
E	e_0	e_1	e_2	.....	e_n
$pre(E)$	nil	e_0	e_1	.....	e_{n-1}

TAB. 3.1 – Opérateur de précédence

L'opérateur *d'initialisation "suivi de*" ou "*init*", noté " $\rightarrow$ ", a la sémantique suivante : si E et F sont des expressions dénotant respectivement les séquences de valeurs $(e_0, e_1, ..., e_n, ...)$ et $(f_0, f_1, ..., f_n, ...)$ alors l'expression $E \rightarrow F$ dénote la séquence des valeurs $(e_0, f_1, ..., f_n, ...)$. Autrement dit, l'opérateur $\rightarrow$ permet de définir la valeur prise par une expression à l'instant initial $(t = 0)$.

	t_0	t_1	t_2	.....	t_n
E	e_0	e_1	e_2	.....	e_n
F	f_0	f_1	f_2	.....	f_n
$E \rightarrow F$	e_0	f_1	f_2	.....	f_n

TAB. 3.2 – Opérateur d'initialisation

```
node n(i1 :t1 ;i2 :t2...)
returns (o1 :tt1 ;o2 :tt2 ;....) ;
var l1 :tl1 ;....
  -- déclaration des variables locales
let
  -- équations
  o=f(i,l) ;
tel ;
```

FIG. 3.2 – Structure d'un noeud LUSTRE

3.3.3 Structure d'un programme LUSTRE

Le langage LUSTRE permet une programmation modulaire. Un module (programme ou sous-programme) est communément appelé *noeud*. Un noeud LUSTRE spécifie une relation entre des variables (flots) d'entrée et des variables (flots) de sortie. Comme le montre la figure 3.2, l'entête d'un noeud LUSTRE comprend :

- la liste des flots d'entrée i_1 de type t_1, i_2 de type t_2, ..., i_m de type t_m
- la liste des flots de sortie o_1 de type tt_1, o_2 de type tt_2, ..., o_n de type tt_n.

Le corps du noeud est formé d'un ensemble d'équations $o = f(i, l)$ et éventuellement d'un ensemble d'assertions et de la déclaration d'un ensemble de flots locaux (variables locales l_1 de type tl_1, l_2 de type tl_2, ..., l_k de type tl_k)

Un noeud LUSTRE manipule des variables (entrées, sorties, locales) qui prennent une nouvelle valeur à chaque instant. Chaque variable de flot (locale ou de sortie) est définie par une seule équation ; elle apparaît en membre gauche de celle-ci. Le membre droit contient son expression de définition à chaque cycle. Cette expression peut être une fonction des valeurs passées de la variable et des valeurs présentes ou passées des autres variables.

L'équation $x = E$ définit la variable x comme identique à l'expression E. En d'autres termes, les deux membres ont la même séquence de valeurs. Le principe de substitution du langage stipule que toute occurrence de x peut être substituée par E et vice et versa. Par conséquent, les équations peuvent être écrites dans n'importe quel ordre, et l'introduction de nouvelles variables (flots locaux) pour renommer des sous-expressions n'a aucune incidence sur la logique du programme.

```
node Never (X :bool) returns (never_A :bool)
let
  never_A = not A->(not A and pre(never_A)) ;
tel ;
```

FIG. 3.3 – Exemple d'un programme LUSTRE

L'exécution d'un noeud se produit quand tous ses opérandes (les données en entrée) sont disponibles. Ainsi, les fonctions sont exécutées dans un ordre résultant seulement des dépendances des données.

Exemple 3.3.1 Un exemple d'un programme est donné en Fig. 3.3. Il s'agit d'un noeud dont la sortie (booléenne) prend la valeur *true* si et seulement si l'entrée (booléenne également) n'a jamais été vraie (*true*) depuis le début de l'exécution. Par exemple, à la séquence de valeurs d'entrée $(false, false, false, true, false)$, le programme associe la séquence de valeurs de sortie $(true, true, true, false, false)$.

Examinons l'équation qui définit la variable $never\_A$. Cette variable est définie à l'aide des opérateurs " $\rightarrow$ " et "*pre*". L'opérateur "*pre*" permet d'accéder à la valeur du flot opérande au cycle précédent. Ainsi, $never\_A$ est définie récursivement sur un temps discret (dont les instants sont des cycles).

LUSTRE permet également d'imposer des propriétés invariantes sur les variables du programme à l'aide de la directive *assert* suivie d'une expression booléenne invariante, appelée *assertion*. L'un des buts de l'utilisation des assertions est de donner des directives au compilateur afin d'optimiser le code lorsque le programme possède quelques propriétés connues. Par exemple, l'assertion 3.1 signifie que les deux événements en entrée représentés par deux variables booléennes x et y n'apparaissent jamais en même temps.

$$assert\ not(x\ and\ y) \tag{3.1}$$

Algorithm 1 Implémentation d'un programme LUSTRE

```
var E, S, M ;
M := m0 ;
proc P_step() ;
foreach step do
   read(E) ;
   P_step() ;
   write(S) ;
end foreach
```

De même, l'assertion 3.2 exprime le fait qu'un événement x n'apparaît jamais deux fois de suite. Outre leur usage dans l'optimisation du code, les assertions jouent un rôle important dans la vérification de programmes LUSTRE.

$$assert\ (true \rightarrow not(x\ and\ pre(x)) \tag{3.2}$$

3.3.4 Compilation d'un programme LUSTRE

La compilation des programmes LUSTRE consiste à passer d'une description parallèle à un programme séquentiel simple. L'implémentation d'un programme réactif synchrone avec un ensemble d'entrées E et un ensemble de sorties S est une boucle simple dont le squelette est donné dans l'algorithme 1.

Le rôle du compilateur consiste à :

1. identifier la mémoire initiale m0,
2. fournir le corps de la boucle (procédure P _step())
3. implémenter de manière efficace cette procédure.

Compilation en boucle simple

L'idée de la compilation en boucle simple est de remplacer toutes les définitions (équations) par des affectations. Le programme obtenu par traduction est un programme séquentiel dans lequel les opérateurs classiques du langage sont traduits de manière triviale et les opérateurs temporels "*pre*" et " $\rightarrow$ " sont traduits par des

variables de mémoire. Le programme C correspondant à la compilation du noeud "never" est donné en annexe.

Compilation en automate de contrôle

L'automate de contrôle associé à un noeud LUSTRE dénote un modèle fini de comportements. La construction de cet automate est basée sur une idée simple consistant à associer à une configuration de la mémoire un état. Chaque état de l'automate représente l'historique de ce qui s'est passé avant d'atteindre cet état. La structure obtenue est un automate de MEALY [24] dans laquelle :

- Les valeurs des entrées et de sorties figurent sur les transitions,
- Un état est composé d'un ensemble de variables d'état sv_i.
- Une variable d'état sv_i est créée pour chaque expression $pre\ E$ du programme ; et sert à conserver la valeur précédente de l'expression E.

Par exemple, l'automate de contrôle (voir Fig. 3.4) associé au noeud "never" de l'exemple 3.3 comprend deux variables d'état $sv_0 = true \rightarrow false$ et $sv_1 = never\_A$.

Du point de vue industriel, la compilation en boucle simple est la solution la plus viable. En dépit du fait qu'elle implique une exécution moins rapide qu'un automate, son avantage est que la taille du code obtenu est linéaire en fonction du nombre d'expressions LUSTRE. Cependant, l'intérêt de compiler en automate est, surtout d'offrir un modèle pour la validation et la vérification.

3.4 Réseau d'opérateurs

Une représentation graphique naturelle de la structure d'un programme LUSTRE est le réseau d'opérateurs. Un réseau d'opérateurs est un graphe orienté constitué de noeuds inter-connectés par des arcs. **Les noeuds**, représentant des unités fonctionnelles, dénotent soit les opérations habituelles associées aux types de base du langage soit des noeuds LUSTRE (ensemble d'opérations réunies dans un module). **Les arcs** modélisent les liens entre les noeuds (opérateurs), les entrées du programme et ses sorties.

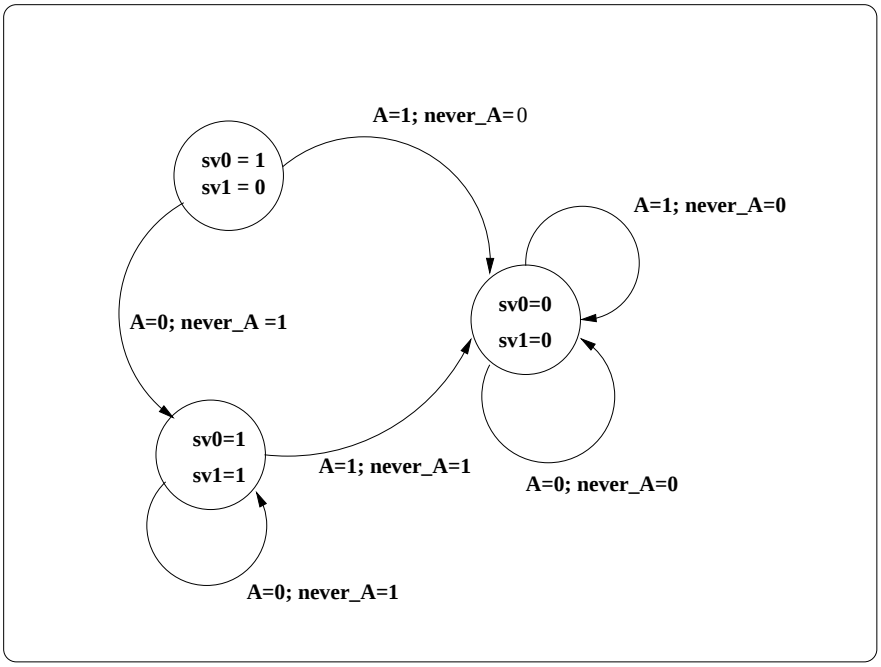

FIG. 3.4 – Automate de contrôle du noeud **never**

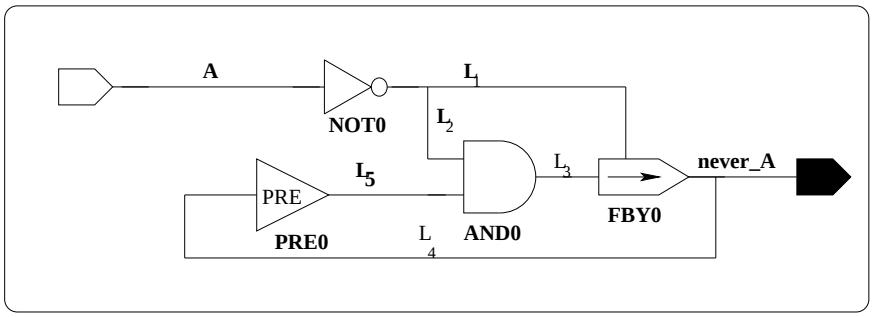

FIG. 3.5 – Réseau d'opérateurs du noeud Never

Le réseau d'opérateurs (cf. figure 3.5) du noeud Never ci-dessus comprend deux opérateurs booléens *not* et *and* et deux opérateurs temporels *pre* et fby. Il comprend également un arc d'entrée A, un arc de sortie $never\_A$ et cinq arcs locaux L_1, L_2, L_3, L_4 et L_5 tous booléens.

3.5 Validation des logiciels à base de LUSTRE

Le langage LUSTRE est utilisé dans des programmes de contrôle/commande réactifs, principalement pour des applications de production de l'énergie électrique et d'avionique. Le degré de criticité de ce type de logiciels nécessite un niveau très

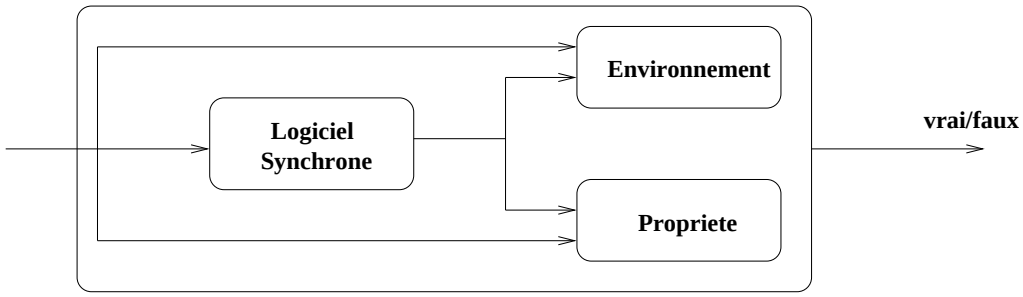

FIG. 3.6 – Schéma de la preuve d'une propriété

élevé de confiance. Les phases de vérification/validation sont donc très développées et leur outillage est indispensable.

3.5.1 Preuve formelle de propriétés

LUSTRE peut servir à la fois de langage de programmation et de logique temporelle du passé. Il est ainsi utilisable pour décrire le comportement du logiciel ainsi que pour l'expression de certaines propriétés invariantes que celui-ci doit satisfaire. La nature formelle du langage offre un cadre permettant une expression formelle et uniforme des différentes spécifications liées à un logiciel. Les premiers travaux ont porté sur la preuve de propriétés [20]. Ces travaux ont mis en évidence un principe de description de logiciels synchrones par trois types de spécifications :

- Une *spécification fonctionnelle*, un noeud LUSTRE décrivant le comportement du logiciel en définissant de le calcul exact des sorties à partir de ses entrées.
- Une *spécification de l'environnement* du logiciel, un ensemble de propriétés invariantes caractérisant l'environnement externe du logiciel.
- Une *spécification de propriétés invariantes dites de sûreté* exprimant l'absence de comportements particulièrement indésirables.

Afin de prouver automatiquement que le programme (noeud LUSTRE) satisfait la propriété, LESAR [50], un outil de model-checking, construit un modèle fini (une machine d'états finis) englobant le programme lui-même, les contraintes d'environnement et la propriété à prouver (voir Fig. 3.6). L'automate est parcouru exhaustivement afin de montrer que la propriété est satisfaite dans tous les états du modèle.

Les propriétés de sûreté sont, en général, des relations simples entre les entrées et les sorties du programme. De ce fait, les modèles manipulés par la preuve restent souvent d'une taille raisonnable. Toutefois, il n'est pas rare que la preuve se heurte au problème classique des techniques basées sur la preuve formelle, à savoir l'explosion combinatoire du nombre d'états du modèle. En effet, ce nombre devient prohibitif au fur et à mesure que la complexité et la taille du programme et des propriétés à vérifier augmentent. Des solutions de compression de modèles basées sur la manipulation symbolique peuvent y remédier mais rendent les traitements plus longs.

Cependant, et comme toutes les techniques basées sur la preuve formelle, l'inconvénient principal reste l'explosion combinatoire vite atteinte pour des programmes de taille importante. La preuve formelle des programmes LUSTRE vise exclusivement à montrer la satisfaction de propriétés considérées mais elle ne permet pas d'affirmer que le logiciel répond réellement aux besoins et ce pour deux raisons [47] :

- L'absence de validation de spécifications, car les propriétés peuvent être insuffisantes ou même fausses. En cas de l'échec de la preuve, il est difficile de décider si nous sommes en présence d'un défaut du logiciel ou d'un défaut dans l'expression des propriétés. Réciproquement, une preuve réussie ne montre que la satisfaction de propriétés dont on n'a aucun moyen d'assurer la pertinence. Ce point de vue est conforté par la définition donnée dans [30], une défaillance se définit par rapport à la fonction du système et non par rapport sa spécification.
- La preuve s'intéresse uniquement aux propriétés de sûreté. Or certains défauts du logiciel n'ont pas d'impact sur la préservation de ces propriétés et passeront, de ce fait, inaperçus.

Pour ces deux raisons, d'autres moyens de validation/vérification ayant pour but de mettre en évidence les défauts du logiciel ont été explorés et plus précisément le test logiciel.

3.5.2 Validation par le test logiciel

Le test est un moyen de validation qui, contrairement à la preuve, a pour but de mettre en évidence les défauts du logiciel. Deux des principales problématiques relatives au test sont d'une part la construction des données de test (critère de sélection, génération automatique,...) et d'autre part l'évaluation de la qualité des données de test (critères de couverture, critère d'arrêt,...). De nombreuses investigations ont été conduites durant ces dix dernières années sur la construction des données de test pour les programmes synchrones. En particulier, des techniques de génération automatique des données de test à la fois du type "boîte noire" [15,44,46,52] que du type "boîte blanche" [33] ont vu le jour. Cependant, peu de travaux ont été consacrés à la couverture [35] et encore moins aux critères d'arrêt.

3.6 Génération de données de test

Les techniques de génération de données de test reposent sur une description du programme sous test. Selon le niveau de connaissance qu'on a du programme sous test, ces techniques se divisent en deux catégories : les techniques de génération à base de test boîte noire, et les techniques de génération à base de test boîte blanche. Une description binaire (exécutable) du programme sous test est suffisante pour générer des données de tests dans le premier cas. En revanche, le code LUSTRE du programme sous test est nécessaire dans le second type de génération.

Dans ce qui suit, nous rappelons de manière succinte les principes de chacune de ces techniques.

3.6.1 Génération pour le test boîte noire

Principes de génération

Les techniques de test du type "boîte noire" visent à détecter la manifestation de défauts présents dans le logiciel sous test. Elles s'appuient sur des modèles analogues à ceux utilisés par la preuve (spécification du programme sous test, spécification

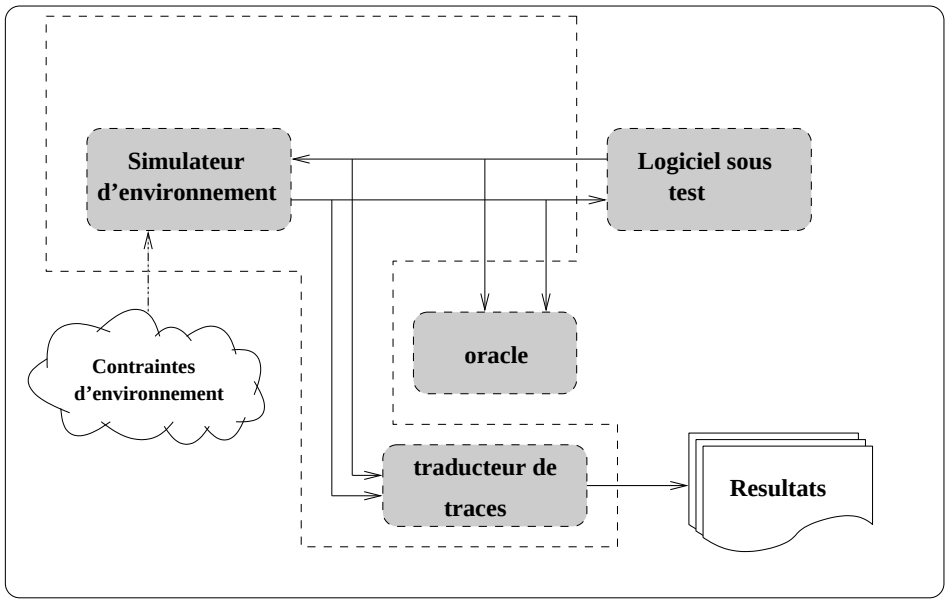

FIG. 3.7 – Architecture de Lutess

d'environnement et spécification des propriétés de sûreté). La génération est basée sur le principe de génération aléatoire sous contraintes, les contraintes étant la spécification de l'environnement ou des propriétés de sûreté. L'objectif des contraintes d'environnement est d'exclure les comportements irréalistes du logiciel ce qui réduit considérablement l'espace des états à explorer. Des comportements irréalistes correspondent, par exemple, à des combinaisons incompatibles des entrées.

La génération aléatoire sous contraintes consiste en la construction d'un programme animant la spécification de l'environnement, appelé *générateur aléatoire contraint* dont le rôle est de simuler le rôle de l'environnement. Le générateur (une machine d'états finis) produit des données de manière dynamique. Il interagit avec le programme sous test, en lui fournissant des données simulant les entrées conformes à la spécification de l'environnement. Le programme sous test calcule une sortie et la renvoie au générateur qui calcule l'entrée suivante et ainsi de suite. Les entrées que le générateur contraint produit sont choisies parmi toutes celles qui respectent les contraintes d'environnement de manière aléatoire.

Pour choisir parmi toutes les données valides (celles qui sont conformes à la spécification de l'environnement), les entrées à envoyer au logiciel sous test, plusieurs stratégies de sélection ont été développées et implémentées dans un outil de généra-

tion appelé LUTESS. Comme le montre la figure 3.7, l'architecture de LUTESS est un harnais constitué du logiciel sous test, du simulateur d'environnement, d'un oracle et d'un traducteur de traces.

LUTESS s'appuie sur le principe de génération aléatoire sous contraintes. Il permet d'engendrer des données de test par :

- génération aléatoire avec tirage équiprobable [46, 48],
- génération guidée par les propriétés de sûreté [26, 46, 48, 57],
- génération guidée par les profils opérationnels [14, 46],
- et la génération guidée par les schémas comportementaux [64].

Génération aléatoire par tirage équiprobable

C'est la technique de génération la plus élémentaire. Dans cette technique, toutes les entrées qui satisfont les contraintes d'environnement ont la même probabilité d'être choisies. La technique consiste, donc, à attribuer la même probabilité de tirage à toutes les entrées valides et à en choisir une de façon aléatoire.

Génération guidée par les profils opérationnels

Un profil opérationnel est un moyen de rendre plus ou moins probable l'occurrence d'un événement selon une condition donnée. La stratégie de génération guidée par les profils opérationnels, intégrée à LUTESS [14, 46], permet de remplacer la sélection équiprobable des vecteurs d'entrée par une sélection respectant les probabilités spécifiées dans les profils. On trouve le même type de génération dans l'outil LURETTE [52] dont les contraintes peuvent porter à la fois sur des données de type booléen qu'entier.

Génération guidée par des schémas comportementaux

Un schéma comportemental est une description du comportement du logiciel qui permet au testeur de guider ce dernier dans une direction donnée. La génération guidée par des schémas comportementaux, intégrée dans LUTESS [64], permet à

celui-ci d'engendrer des données qui mènent à la réalisation de toutes les étapes du schéma comportemental.

Génération guidée par des propriétés de sûreté

Une propriété de sûreté, associée à un programme LUSTRE, est une expression booléenne qui doit être toujours vraie. La génération guidée par les propriétés de sûreté, introduite dans [46] et largement développée dans [57] a pour objectif de placer le logiciel dans une situation où il est susceptible de violer une propriété de sûreté.

3.6.2 Génération pour le test boîte blanche

Dans ce domaine, la principale technique, représentée par L'outil GATEL [33,34], repose sur une interprétation des constructions du langage LUSTRE sous la forme de contraintes sur des variables booléennes ou des variables à valeurs dans des intervalles d'entiers. La génération des données de test consiste à résoudre ces contraintes à l'aide de la programmation logique par contraintes (PLC).

GATEL s'appuie sur le code LUSTRE du programme sous test et sur une description de son environnement. Son objectif est d'engendrer des séquences de test (entrées et sorties) à partir d'objectifs de test formulés par l'utilisateur et si nécessaire de critères de couverture structurels construits interactivement. Cette seconde possibilité est utilisée dans le but d'assurer une couverture structurelle de l'objectif de test (voir section 3.7.5)

Un objectif de test peut être la traduction d'une propriété à tester, ou bien un sous-domaine du domaine de définition du programme sous test, caractérisé par un prédicat du chemin de calcul. Les propriétés considérées par GATEL peuvent être soit des propriétés invariantes exprimées par la directive *assert* ou des propriétés qui doivent être vérifiées au moins sur un cycle et exprimées par la directive *reach*.

Le système de contraintes construit par GATEL est dérivé d'un objectif de test, de la description de l'environnement (assertions de chaque cycle, propriétés apparaissant dans la directive reach au dernier cycle) et des définitions des variables

nécessaires issues du programme LUSTRE.

La résolution procède par éliminations successives de toutes les contraintes en construisant un passé compatible qui correspond à une exploration en arrière de certaines variables de sortie.

La résolution du systèmes de contraintes fournit une séquence de tests. Afin d'obtenir plusieurs séquences de test, GATEL propose de décomposer les domaines de valeurs des variables d'entrée en sous-domaines. Ceci est obtenu par dépliage des opérateurs LUSTRE (voir section 3.7.5).

3.7 De la couverture dans le test des logiciels synchrones

Les techniques de test présentées dans la section précédente ont comme objectif d'engendrer des données de test afin de tester un programme LUSTRE. Cependant, dans un processus de test, il ne suffit pas de générer des données de test. En effet, il faudrait fournir des moyens qui permettent d'évaluer les jeux de tests construits et à terme de décider de l'arrêt du test. Habituellement, les techniques de couverture permettent de répondre à de telles préoccupations. Cette couverture peut être fonctionnelle (couverture de fonctions, de propriétés) ou structurelle (code source du programme ou certains de ces modèles).

3.7.1 Couverture statistique de l'automate de contrôle

Dans [35, 54], le test statistique a été utilisé pour mesurer la couverture de l'automate de contrôle. L'objectif de cette technique est d'engendrer des données de test qui couvrent des branches de l'automate. L'approche proposée ne définit pas de critère de couverture structurelle sur le langage LUSTRE à proprement parlé, mais sur l'automate de contrôle. Plus précisément, le programme est transformé en un modèle stochastique de son comportement en associant à chaque transition une probabilité : une séquence d'entrées est ensuite calculée de telle sorte que la probabilité

de couverture des états, des transitions ou des séquences de deux transitions soit supérieure à une valeur seuil fixée au préalable.

3.7.2 Couverture des réseaux d'opérateurs

Dans le cadre de programmes écrits en langages impératifs séquentiels, des techniques appropriées, dites structurelles, basées sur l'analyse du code source du programme ont été proposées [42]. Des erreurs d'implantation peuvent également être commises quand Lustre est utilisé pour la programmation d'un logiciel réactif. Il semble donc naturel que des techniques de test structurel soient utilisées pour tester un tel logiciel. La majorité des techniques structurelles de test pour langages impératifs utilisent comme modèle du programme un graphe de contrôle.

Cette forme de représentation n'est pas adaptée à un langage flot de données comme Lustre. Un programme Lustre est plutôt représenté par un réseau d'opérateurs. En effet, un tel programme, n'est qu'un ensemble non ordonné d'équations définissant des relations invariantes entre les entrées et les sorties. En conséquence, un réseau d'opérateurs reflète plus naturellement la structure du programme.

Un premier travail a été proposé dans [44]. L'objectif de ce travail a consisté à définir sur cette représentation des critères analogues à ceux utilisés pour les langages impératifs. De la même manière que les critères des programmes impératifs concernent la couverture des instructions, des branches et des chemins, les critères proposés concernent la couverture des opérateurs, des arcs et des chemins. Un chemin est défini comme une suite finie d'arcs contigus du réseau d'opérateurs où le premier arc est une entrée du réseau et le dernier est une sortie du réseau.

De manière informelle, une séquence d'entrée satisfait le critère de couverture d'opérateurs si, pendant leur traitement par le logiciel, au moins un arc de sortie de tout opérateur appartient à l'un des chemins traversés. Cela signifie que l'opérateur a été activé et que ses résultats ont été utilisés pour le calcul de la sortie du chemin. De manière similaire, la couverture des arcs est satisfaite si tous les arcs du réseau d'opérateurs font partie des chemins traversés. La couverture des chemins est impossible pour des séquences d'entrée dont la longueur n'est pas fixée. Une

définition formelle de ces critères a été également proposée. Dans cette définition, un chemin est muni d'un prédicat dit prédicat de chemin. Un prédicat de chemin est la condition sous laquelle le chemin est traversé par le flot de données.

3.7.3 Couverture du code C engendré

Comme nous venons de le voir ci-dessus (section 3.3.4), un programme LUSTRE peut être compilé en un programme écrit dans le langage C. Sur ce code, on peut envisager d'appliquer toutes les techniques de couverture structurelle présentées dans le chapitre 2. En particulier, l'outil SCADE Suite, grâce au module MTC (Model Test Coverage), permet de mesurer la couverture structurelle du code C engendré selon la norme DO-178B (voir section 2.6).

3.7.4 Couvertures fonctionnelle et mixte

Dans les techniques de test boîte noire, les génération aléatoire et statistique peuvent être interprétées comme visant une couverture fonctionnelle du logiciel sous test. Dans le test aléatoire, les contraintes d'environnement permettent de décomposer l'ensemble des données d'entrée en sous-ensembles correspondant chacun à une fonctionnalité. Un jeu de tests qui satisfait ces contraintes est un jeu de tests qui couvre fonctionnellement le programme sous test.

Dans le test guidé par les profils opérationnels, ces derniers correspondent à une décomposition fonctionnelle du programme sous test selon divers usages. Un jeu de tests conforme aux profils opérationnels est un jeu de tests qui satisfait la couverture des fonctionnalités du logiciel relatifs aux profils.

Le test basé sur les propriétés de sûreté [57] vise la couverture de certains comportements menant à des états suspects.

3.7.5 Couverture structurelle d'un objectif de test

Pour engendrer plusieurs séquences de test, GATEL s'appuie sur des techniques de découpage d'un domaine de valeurs en sous-domaines en dépliant les opérateurs

LUSTRE.

Par exemple, si le système de contraintes contient la contrainte *S=if Cond then ExpThen else ExpElse* où *Cond* est une variable booléenne, GATEL peut dériver deux sous-domaines par dépliage du $if - then - else$. Le premier sous-domaine contient tous les tests tels que *Cond* est vraie tandis que le second contient les tests tels que *Cond* est fausse. Ces deux sous-domaines sont définis par les deux systèmes de contraintes obtenus par propagation des valuations possibles de *Cond*. Ils peuvent à leur tour être découpés par dépliage d'un des opérateurs apparaissant dans leurs contraintes de définition.

A chaque étape de découpage, GATEL indique les différents opérateurs dépliables et l'utilisateur peut choisir la nature et le nombre des découpages. Les autres opérateurs booléens de LUSTRE peuvent être à leur tour dépliés, en respectant la sémantique de chacun d'eux. Par exemple, l'opérateur *not* supporte deux découpages possibles (*vrai* et $faux$). Pour les autres opérateurs booléens, il existe plusieurs possibilités qui varient selon la finesse de décomposition. Par exemple, l'opérateur *and* dans l'expression $A = Exp1\ and\ Exp2$ supporte trois types de découpage :

- Découpage séquentiel : Deux cas correspondants aux valuations de A.
- Découpage paresseux : Trois cas correspondent à trois niveaux de valuation possibles :
  - $A = true$ et $Exp1 = true$ et $Exp2 = true$
  - $A = false$ et $Exp1 = false$
  - $A = false$ et $Exp2 = false$
- Découpage normal : Quatre cas correspondant à la table de vérité.

Pour l'opérateur temporel $\rightarrow$, deux statuts (*initial* et *non_ initial*) sont associés au cycle en cours. D'autres règles de découpage sont appliquées aux autres opérateurs (voir [33]).

GATEL permet d'appliquer une décomposition en sous-domaines à partir des contraintes définissant une sortie sélectionnée. Pour cette sortie, GATEL construit un *arbre de test* où les noeuds correspondent aux différents découpages possibles des opérateurs intervenant dans l'expression définissant la sortie. Un *chemin de calcul*

est un parcours de cet arbre de la racine à une feuille. Un *prédicat de chemin* est la conjonction des valeurs portées par les noeuds.

L'utilisateur peut choisir de ne déplier que certains opérateurs. Sur l'arbre de test résultant, il doit décider du niveau de couverture des chemins de calcul : tous les chemins ou certains sélectionnés interactivement. Ainsi, GATEL permet une couverture structurelle des expressions intervenant dans un objectif de test.

3.7.6 Couverture de modèles de fautes dans les circuits digitaux

Dans les circuits digitaux, un modèle de fautes correspond à une caractérisation des comportements fautifs du circuit lors de sa fabrication. A partir de ce modèle, le concepteur peut prévoir les conséquences d'une faute particulière.

Le modèle de fautes le plus utilisé dans la pratique est le modèle de fautes par collages [2]. Dans ce modèle, un signal du circuit est collé de manière permanente à une valeur logique fixe (0 ou 1) indépendamment des entrées du circuit. Dans ce modèle, le testeur suppose que tout défaut physique a pour effet que le circuit se comporte comme si un et un seul signal du circuit était collé exclusivement à 0 ou 1. Avec cette hypothèse, le nombre total de circuits fautifs est dénombrable, et égal à 2N où N est le nombre de signaux dans le circuit. Le collage est un modèle de fautes dit *structurel* car il est défini sur un modèle structurel du circuit (portes logiques).

Les outils de génération dits ATPG (Automatic Test-Pattern Generators) tentent de trouver des séquences d'entrées qui, une fois appliquées sur un circuit digital, permettent aux testeurs de distinguer le comportement correct du comportement incorrect causé par une faute particulière. La génération consiste à trouver un vecteur de test qui détecte un collage particulier d'un signal du circuit.

La couverture de fautes se réfère au pourcentage de fautes qui peuvent être détectées durant le test d'un circuit. Le taux de couverture est la métrique qui permet de mesurer l'efficacité d'un jeu de tests, en calculant le rapport du nombre de fautes détectées par le jeu de tests au nombre de fautes détectables (modèle). La

simulation de fautes permet de calculer ce taux en construisant progressivement une liste de tous les circuits fautifs détectés par au moins un vecteur.

Le test des circuits consiste [38] à injecter des fautes dans des signaux par la technique du collage. Lorsque la faute modifie le comportement du circuit, le changement provoque ce qu'on appelle un effet de faute. Les effets des fautes traversent le circuit en en provoquant d'autres. Ce phénomène est appelé propagation de fautes. Le principe de *propagation* est utilisé pour tracer l'effet d'une faute sur le circuit en *sensibilisant* un chemin dans le circuit. Afin de propager un signal d'une entrée à une sortie à travers un circuit, les autres entrées du circuit sont fixées à une valeur qui permet d'observer l'effet du signal propagé. Ainsi, pour propager une valeur d'un signal a_1 à travers une porte AND (respectivement OR) l'autre signal d'entrée a_2 est mis à 1 (respectivement 0).

3.8 Synthèse et Conclusion

Nous venons de voir que les principaux travaux sur le test des programmes synchrones écrits en Lustre se sont intéressés aux aspects de génération (Lutess, Lurette, Gatel,...). Cependant, quelques travaux ont été explicitement consacrés aux aspects de couverture, en particulier la couverture en tant que critère sélection.

Dans le chapitre suivant, nous proposons une approche pour la couverture des programmes Lustre. L'approche consiste à définir des critères de couverture spécifiques au paradigme flot de données synchrone.

Chapitre 4

Une approche pour la couverture structurelle des programmes LUSTRE

4.1 Introduction

L'une des problématiques du test est de définir un critère qui permet d'affirmer qu'un programme est "bien testé". Un tel critère doit fournir des moyens pour mesurer la qualité d'un jeu de tests et son aptitude à détecter les défauts dans un programme. Dans le cas des applications synchrones à base de LUSTRE, nous avons vu au chapitre 3 qu'il existe de nombreux travaux relatifs à la génération de données de test visant la couverture des séquences d'entrées ou une bonne couverture des comportements [35, 57].

Dans les techniques de génération utilisées par LUTESS [14, 46, 64], une couverture du domaine constitué des séquences d'entrée peut être recherchée par simulation aléatoire du comportement de l'environnement. Dans sa version de base, le générateur qui simule l'environnement effectue un tirage complètement aléatoire sur toutes les séquences d'entrées qui respectent les contraintes d'environnement. Les séquences d'entrées générées couvrent, de ce fait, l'ensemble des séquences possibles. La stratégie de sélection guidée par des profils opérationnels vise, d'une certaine façon, une couverture statistique basée sur les usages du programme sous test. De même pour la génération guidée par les schémas comportementaux [64].

Dans une extension récente de LUTESS [57], une approche est proposée afin d'assurer la couverture de certains comportements menant à des états dits *suspects*. Considérer "la couverture des états suspects" comme un critère pour l'arrêt de test suppose une connaissance préalable de ces états et une analyse fine de l'accessibilité. Or pour des automates relativement grands, cette analyse devient difficile même si des solutions de réduction d'automates peuvent y remédier [57].

Ces techniques sont bien adaptées à la recherche des défauts consécutifs à une mauvaise compréhension de la spécification des propriétés de sûreté ou des contraintes d'environnement. Or, il est fréquent que l'on s'intéresse à la détection des défauts d'implantation. Ces derniers sont liés au langage de programmation utilisé et peuvent être très variés : utilisation d'un mauvais opérateur, constante ou variable, inversion de condition dans une expression conditionnelle.

Dans les programmes impératifs séquentiels, les techniques de couverture structurelle répondent souvent à ces préoccupations. Les critères de couverture des instructions, de couverture des branches ou de couverture des LCSAJ offrent des moyens mesurables et toujours utilisables pour la couverture du graphe de flot de contrôle quel que soit la complexité de ce dernier. Ces critères ne sont pas immédiatement "transposables" sur des programmes de nature "flot de données" comme les programmes LUSTRE. Et ce, même si dans tout programme LUSTRE il existe un graphe de contrôle (souvent très rudimentaire) pour chaque équation (i.e. instruction), la couverture des branches de cet ensemble de graphes de contrôle s'effectuant généralement en quelques pas de tests. Ce qui ne semble pas un critère pertinent pour l'arrêt de test.

4.2 Objectifs

Dans ce travail, nous proposons une approche pour mesurer la couverture des programmes synchrones écrits en LUSTRE. L'approche est basée sur des critères de couverture adaptés au paradigme flot de données synchrone. A travers cette approche, notre objectif est double.

1. Offrir aux testeurs des critères précis et pertinents pour le paradigme synchrone, leur permettant d'assurer un niveau de couverture satisfaisant d'un composant logiciel synchrone (code ou spécification).
2. Fournir un moyen simple et toujours utilisable pour évaluer la progression du processus de test.

Afin d'atteindre ce double objectif, nous nous sommes posés les trois questions suivantes :

1. *Quel modèle de couverture faut-il utiliser ?*
2. *Quelles sont les réflexions qui guident et orientent la définition des critères ?*
3. *Quels critères sont adéquats pour couvrir ce modèle ?*

4.3 Modèle de couverture

Les principales techniques de test structurel que nous avons survolées au chapitre 2 sont conçues pour des programmes décrits dans des langages impératifs séquentiels. Les critères de couverture proposés sont définis sur le graphe de contrôle, modélisation naturelle de la structure des programmes. Cette représentation n'est pas adaptée aux programmes Lustre qui, contrairement aux langages impératifs, définissent un traitement du flot de données qui les traverse.

4.3.1 Flot de données et réseau d'opérateurs

Un programme Lustre définit une fonction entre des flots d'entrée et des flots de sortie (voir Fig. 4.1). Cette fonction est constituée d'un ensemble de fonctions partielles dont chacune correspond à une ou plusieurs équations. Chaque équation $o_i = f(i_0, i_1, ..., i_n)$ est, schématiquement, représentée par une unité fonctionnelle dont l'exécution, à un instant t est conditionnée par la présence à son entrée de toutes les valeurs des entrées i_k à cet instant.

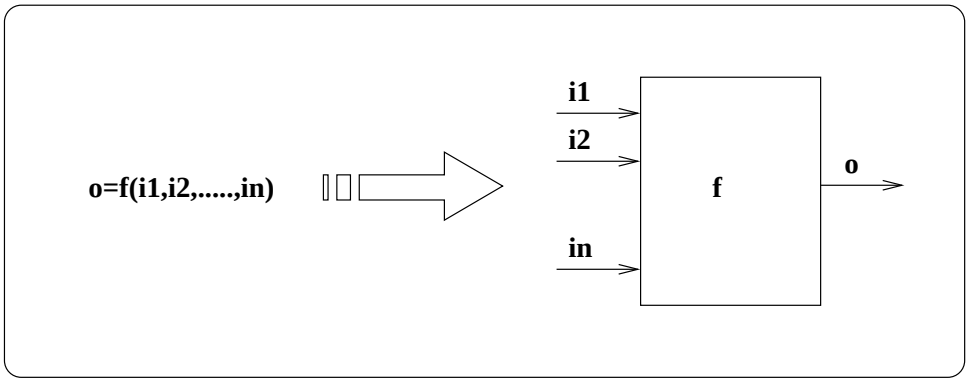

FIG. 4.1 – Équation <=>Opérateur

Un programme LUSTRE ressemble, donc, graphiquement à une structure où les différentes unités fonctionnelles sont inter-connectées par des arcs portant chacun un flot de données.

4.3.2 Autres modèles

D'autres modèles de programmes LUSTRE issus de la compilation de celui-ci peuvent être envisagés. Un programme LUSTRE est compilé soit en un automate d'états finis ou bien en un programme séquentiel C (cf section 3.3.4). Sur l'automate de contrôle, des critères de couverture classiques tels que la couverture des états ou la couverture des transitions, ou même celle des séquences de transitions, peuvent être appliqués. Cette approche a été adoptée par [35] (voir section 3.7.1). Sur le programme C généré, l'idée est d'utiliser le graphe de contrôle du programme C issu de la compilation du programme LUSTRE; c'est le cas de l'outil MTC de l'environnement SCADE Suite (voir section 3.7.3). Ce modèle présente un certain intérêt pratique parce qu'il permet d'utiliser les nombreux outils de test disponibles pour le langage C.

Ces deux modèles présentent un inconvénient commun. En effet, lors de sa compilation, le programme LUSTRE est transformé en un noeud unique aplati (sans appels d'autres noeuds). Il en résulte que le programme C et l'automate engendrés ne reflètent plus la structure du programme initial. De plus, les compilateurs efficaces produisent du code optimisé dont la couverture est difficilement interprétable

en termes du code source.

Dans ce travail, nous proposons des critères de couverture des réseaux d'opérateurs, modélisation naturelle des programmes LUSTRE. En effet, dans le secteur industriel, les concepteurs des systèmes LUSTRE/SCADE manipulent des représentations graphiques (réseaux d'opérateurs) des programmes LUSTRE. Des critères de couverture de ces programmes auront plus d'utilité que les critères de couverture des automates de contrôle et des programmes C générés équivalents.

4.4 Réseau d'opérateurs

Dans la suite de ce document, nous adoptons les définitions suivantes :

4.4.1 Définitions de base

Définition 4.4.1 *Un réseau d'opérateurs est un graphe étiqueté, orienté, multi-entrées, multi-sorties, où les noeuds dénotent des opérateurs et les arcs représentent les flots de données reliant ces opérateurs.*

Noeuds

Un noeud du graphe dénote une unité fonctionnelle transformant des flots d'entrée en flot de sortie. Un noeud est muni d'un identificateur et d'un type. On en distingue trois types : les noeuds d'entrée, les noeuds de sortie et les noeuds internes.

Un noeud d'entrée (respectivement de sortie) est un noeud qui n'a pas de flots en entrée (respectivement en sortie) et dénote un point d'entrée (respectivement de sortie) du graphe. Un noeud interne dénote un opérateur du programme LUSTRE. Les réseaux d'opérateurs que nous utilisons dans le cadre de ce travail utilisent des noeuds internes associés aux opérateurs suivants :

1. *Opérateurs booléens*
   (a) l'opérateur unaire de négation **NOT**
   (b) l'opérateur binaire de conjonction **AND**

(c) l'opérateur binaire de disjonction **OR**

2. *Opérateurs arithmétiques*

   (a) l'opérateur unaire de négation (**MINUS**)

   (b) les opérateurs binaires d'addition (**ADD**), de soustraction (**SUB**), de multiplication (**MUL**), de division (**DIV**) et de reste de la division (**MOD**).

3. *Opérateurs relationnels*

   (a) l'opérateur "*plus grand que*" (**GT**)

   (b) l'opérateur "*plus petit que*" (**LT**),

   (c) l'opérateur "*plus grand que ou égal à* " (**GTE**),

   (d) l'opérateur "*plus petit que ou égal à* " (**LTE**),

   (e) l'opérateur "*égal à* " (**EQ**)

   (f) l'opérateur "*est différent de*" (**NEQ**).

4. *Opérateurs temporels*

   (a) l'opérateur de mémoire (**PRE**)

   (b) l'opérateur d'initialisation (**FBY**).

Arcs

Un arc du graphe modélise un flot de données entre deux noeuds. Il dénote l'occurrence d'une variable dans une expression du programme LUSTRE associé. Formellement, un arc peut être défini comme un fil ayant deux attributs *value* et *label*. Le premier spécifie la valeur du flot qu'il porte et le second est une étiquette qui identifie l'arc. De même, on associe à chaque arc e les deux fonctions $source(e)$ et $destination(e)$ qui donnent respectivement l'ensemble des noeuds sources et l'ensemble des arcs puits.

De manière duale, on associe à chaque noeud *node* les deux fonctions $in(node)$ et $out(node)$ qui renvoient respectivement l'ensemble des arcs d'entrée du noeud et l'ensemble des arcs de sortie du noeud.

On note l'ensemble des arcs d'un graphe par $\mathcal{E} = \mathcal{I} \cup \mathcal{L} \cup \mathcal{O}$ tel que :

- $\mathcal{I}$ dénote l'ensemble des arcs d'entrée,
- $\mathcal{O}$ dénote l'ensemble des arcs de sortie
- $\mathcal{L}$ dénote l'ensemble des arcs internes.

Deux arcs e et s sont dits adjacents si et seulement s'il existe un noeud op tel que $e \in in(op) \ \wedge \ s \in out(op)$

4.4.2 Exemple : Contrôleur d'un climatiseur

Dans la suite de ce chapitre et afin d'illustrer les notions que nous allons introduire, nous utilisons un contrôleur de climatiseur comme exemple.

L'environnement du contrôleur est composé d'un interrupteur, d'un capteur émettant trois signaux de température et d'une soufflerie. Le contrôleur utilise les signaux de température et de position de l'interrupteur pour commander la soufflerie et répondre aux besoins de l'utilisateur ; ces quatre signaux sont décrits ci-dessous :

- **Marche** est *vrai* lorsque le climatiseur est sous tension.
- **TempInf** est *vrai* lorsque la température lue par le capteur est inférieure à la température souhaitée par l'utilisateur.
- **TempOk** est *vrai* lorsque la température lue par le capteur est égale à la température souhaitée par l'utilisateur.
- **TempSup** est *vrai* lorsque la température lue par le capteur est supérieure à la température souhaitée par l'utilisateur.

Le capteur permet à la fois de régler la température souhaitée par l'utilisateur et de mesurer la température ambiante. Le résultat de la comparaison des deux températures aboutit à l'envoi d'un des trois signaux **TempInf**, **TempOk** et **TempSup**. Les commandes qui permettent au contrôleur d'agir sur la soufflerie sont :

- **Arrete** qui permet de mettre la soufflerie hors tension. Il vaut *vrai* lorsque l'interrupteur est sur la position "*off*".
- **Inactif** met le climatiseur en veille
- **Chaud** active la soufflerie pour diffuser de l'air chaud.
- **Froid** active la soufflerie pour diffuser de l'air froid.

Le noeud LUSTRE qui modélise le contrôleur du climatiseur est donné dans la figure

```
node Climatiseur(Marche,TempInf,TempOk,TempSup :bool)
returns (Arrete,Chaud,Inactif,Froid :bool)
let
  Arrete = false->if Marche then not pre Arrete else pre Arrete ;
  Inactif = Arrete and TempOk ;
  Chaud = Arrete and TempInf ;
  Froid = Arrete and TempSup ;
tel ;
```

FIG. 4.2 – Le noeud Climatiseur

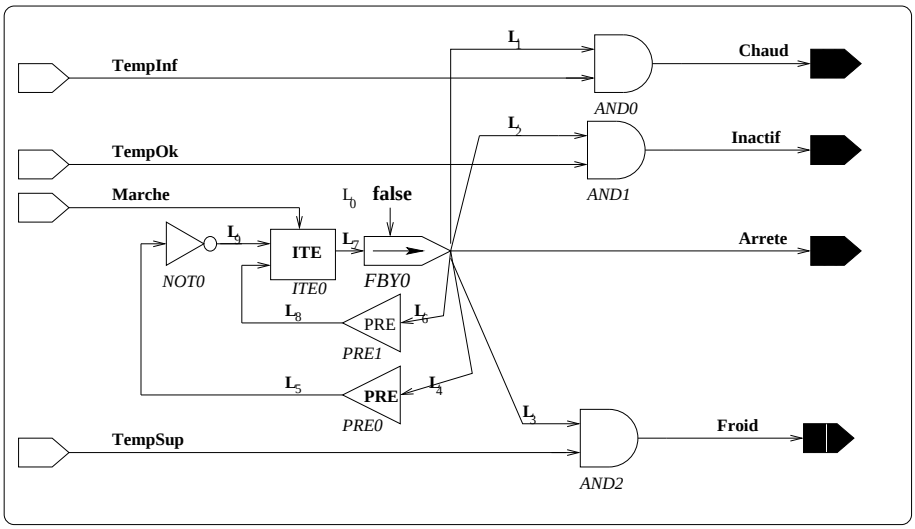

FIG. 4.3 – Réseau d'opérateurs du noeud Climatiseur

4.2.

Comme le montre la figure 4.3, le réseau d'opérateurs du contrôleur est un graphe avec quatre noeuds d'entrée et quatre noeuds de sortie. Le réseau est constitué de huit opérateurs AND0, AND1, AND2, PRE0, PRE1, NOT0, ITE0 et FBY0 et de 18 arcs booléens.

4.4.3 Chemins

Définition 4.4.2 *Soit $n > 1$ un entier positif. Un chemin p, noté $(e_1, ..., e_n)$ est une suite finie d'arcs e_i tels que $\forall i \in [1, n-1]$: e_i et e_{i+1} sont deux arcs adjacents.*

Soit l'entier $n \geq 1$ et soit le chemin $p = (e_1, ..., e_n)$.

Les notations suivantes seront utilisées dans la suite de ce document.

- $length(p)$ dénote la longueur du chemin p.
- $first(p)$ ($last(p)$ respectivement) retourne le premier (respectivement le dernier) arc du chemin p.

Définition 4.4.3 *Longueur d'un chemin*

Soit $p = (e_1, ..., e_n)$ un chemin dans le réseau. La longueur du chemin p, notée length(p), est le nombre d'arcs qui le composent.

Par exemple, le chemin $(Marche, L_7, , L_1, Chaud)$ est un chemin de longueur 4.

L'ensemble de tous les chemins du réseau de longueur n est notée par $P_n = \{p \mid lenght(p) = n\}$. De même, l'ensemble de tous les chemins du réseau dont la longueur est inférieure ou égale à est notée par $\mathcal{P}_n = \bigcup_{k=2}^{k=n} P_k$.

Un chemin p est dit chemin unitaire si et seulement si $length(p) = 2$. Par exemple, les chemins $(TempInf, Chaud)$ et $(Marche, L_7)$ sont des chemins unitaires

Un chemin p' est dit préfixe du chemin $p = (e_1, ...e_{n-1}, e_n)$ si et seulement si $p' = (e_1, ..., e_{n-1})$.

Définition 4.4.4 *Chemin avec mémoire*

- *Un chemin unitaire (e, s) est dit avec mémoire si et seulement s'il existe un noeud N du type pre tel que $e \in in(N)$ et $s \in out(N)$.*
- *Un chemin $p = (e_1, ..., e_n)$ est dit avec mémoire si et seulement si $\exists i \in [1, n-1]$ tel que (e_i, e_{i+1}) est un chemin unitaire avec mémoire.*

Définition 4.4.5 *Cycle dans un chemin*

Soit $p = (e_1, ..., e_n)$ un chemin dans le réseau. On appelle cycles dans le chemin p le nombre $k < n$ de chemins unitaires avec mémoire dans p, on dit que p contient k cycles.

Définition 4.4.6 *Chemin élémentaire*

Un chemin $p = (e_1, ..., e_n)$ est dit élémentaire s'il est sans cycles.

- $p_1 = (Marche, L_7, L_6, L_8, L_7, Arrete)$ est un chemin de longueur 6 ayant un seul cycle.
- $p_2 = (Marche, L_7, L_6, L_8, L_7, L_4, L_5, L_9, L_7, Arrete)$ est un chemin de longueur 10 ayant deux cycles.
- $p_3 = (Marche, L_7, L_2, Inactif)$ est un chemin élémentaire.

Définition 4.4.7 *Types de chemins*

Un chemin p est un chemin complet si et seulement si $first(p) \in \mathcal{I}$ *et* $last(p) \in \mathcal{O}$. *En d'autres termes, s'il relie un noeud d'entrée à un noeud de sortie. Un chemin complet est dit booléen si la valeur de* $first(p)$ *est du type booléen.*

Les chemins p_1, p_2 et p_3 ci-dessus sont tous des chemins complets booléens.

4.5 Notion de condition d'activation

L'analyse du flot de données consiste à identifier les dépendances de données entre deux points du réseau (arcs). Afin de caractériser cette dépendance, nous introduisons la notion d'activation du chemin reliant ces deux arcs. La notion d'activation de chemin illustre la propagation de l'effet de l'arc d'entrée dans l'arc de sortie. Nous donnerons dans la section suivante une définition plus formelle de cette notion. Pour l'instant, nous considérons qu'un chemin est activé si l'entrée a un effet sur la valeur de la sortie. Autrement dit, si tout changement de la valeur de l'entrée provoque un changement de la valeur de la sortie (on dit que la sortie est sensible à l'entrée à travers le chemin).

Partant du fait qu'entre deux arcs du réseau, il peut y avoir plus d'un chemin. La notion d'activation dépend non seulement de l'arc d'entrée et de l'arc de sortie mais aussi du chemin entre les deux. Dans notre contexte, la condition d'activation dénote, donc, une contrainte sous laquelle la valeur d'une entrée est acheminée vers la sortie à travers un chemin. Cette dépendance est construite de manière progressive à partir des différentes dépendances entre les chemins unitaires.

4.5.1 Condition d'activation

Définition 4.5.1 *Soient :*

- $p = (e_1, ...e_{n-1}, e_n)$ *un chemin de longueur* n
- $p' = (e_1, ...e_{n-1})$ *le préfixe de* p.
- *l'opérateur* op *tel que* $in(op) = e_{n-1}$ *et* $out(op) = e_n$

La condition d'activation du chemin p *est une expression booléenne temporelle, notée* $\mathcal{AC}(p)$*, définie comme suit :*

1. *Si* $n = 1$ *alors* $\mathcal{AC}(p) = \ true$.
2. *Sinon,*
   (a) *Si op est un opérateur booléen, relationnel ou conditionnel alors*
      i. $\mathcal{AC}(p) = \ \mathcal{AC}(p')\ and\ \mathcal{OC}(e_{n-1}, e_n)$
      ii. $\mathcal{OC}(e_{n-1}, e_n)$ *est une expression booléenne qui dépend de l'opérateur op.*
   (b) *Si op est un opérateur pre, alors* $\mathcal{AC}(p) = \ false \rightarrow pre(\mathcal{AC}(p'))$
   (c) *Si op est un opérateur* $fby(init, nonInit)$*, et si* p_{init}*,* $p_{nonInit}$ *sont les préfixes associés aux arcs init, et nonInit respectivement, alors :*
      i. $\mathcal{AC}(p) = \ \mathcal{AC}(p') \rightarrow false \quad si\ p' = p_{init}$
      ii. $\mathcal{AC}(p) = \ false \rightarrow AC(p')\ \ si\ p' = p_{nonInit}$

Dans cette définition, la condition d'activation d'un arc est toujours vraie. La condition d'activation d'un chemin de longueur n quelconque est définie de manière récursive selon les opérateurs qui relient les arcs du chemin.

On en distingue trois cas de figure :

1° cas : Opérateurs booléens, relationnels et conditionnels

Dans ce cas, la condition d'activation est une simple conjonction entre la condition d'activation du préfixe et la condition du chemin unitaire (e_{n-1}, e_n) comme le montre la figure 4.4.

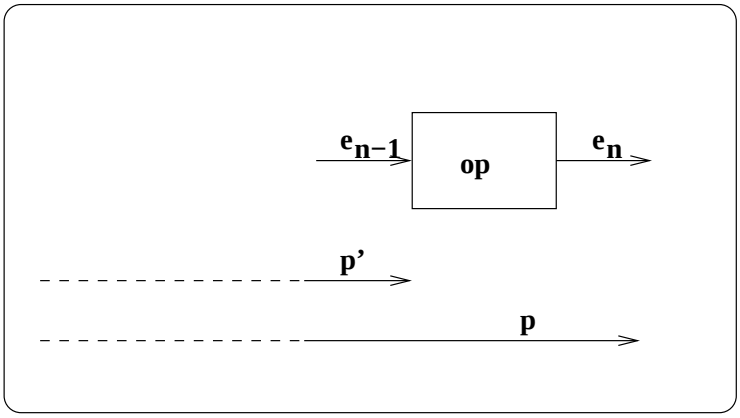

FIG. 4.4 – Construction via un opérateur booléen, relationnel ou conditionnel

L'expression booléenne $\mathcal{OC}(e_{n-1}, e_n)$ associée à un opérateur booléen, relationnel ou conditionnel exprime une dépendance entre l'entrée e_{n-1} et la sortie e_n. Cette expression, appelée condition du chemin unitaire, dépend du type de l'opérateur et est définie formellement comme suit :

Définition 4.5.2 *Condition du chemin unitaire*

Soit le chemin unitaire (e, s) *et l'opérateur op tel que* $e \in in(op)$ *et* $s \in out(op)$. *La condition du chemin unitaire associée à* (e, s) *est une expression booléenne* $\mathcal{OC}(e, s)$ *définie comme suit :*

1. *Si l'opérateur op est un opérateur NOT, alors* $\mathcal{OC}(e, s) = true$
2. *Si l'opérateur op est un opérateur AND tel que* $in(op) = \{e, e'\}$, *alors* $\mathcal{OC}(e, s) = not(e)$ *or* e'
3. *Si l'opérateur op est un opérateur OR tel que* $in(op) = \{e, e'\}$, *alors* $\mathcal{OC}(e, s) = e$ *or* $not(e')$
4. *Si l'opérateur op est un opérateur relationnel, alors* $\mathcal{OC}(e, s) = true$
5. *Si l'opérateur op est un opérateur ITE et si* $in(op) = \{c, e, e'\}$, *alors*
   (a) $\mathcal{OC}(c, s) = true$
   (b) $\mathcal{OC}(e, s) = c$

(c) $\mathcal{OC}(e', s) = not(c)$

La condition du chemin unitaire associé à l'opérateur "**NOT**" signifie que la sortie dépend toujours de l'entrée. Par ailleurs, les conditions des chemins unitaires associés aux opérateurs "**AND**" et "**OR**" sont symétriques, car la sémantique de ces deux opérateurs dans le langage LUSTRE impose une évaluation sans aucune priorité, ni aucun ordre pour les opérandes.

Les conditions de chemins unitaires ci-dessus ont été établies à partir d'une formule traduisant l'intuition énoncée au début de cette section. Par exemple, pour l'opérateur AND, l'expression $not(e)\ or\ e'$ est obtenue de la manière suivante :

$$\mathcal{OC}(e, s) = if\ e\ then\ e'\ else\ true \tag{4.1}$$

Cette équation signifie que l'entrée e a un effet sur la sortie s, à travers l'opérateur AND, si

- e' est vraie lorsque e l'est aussi
- toujours lorsque e est fausse.

L'équation 4.1 est re-écrite comme suit

$$\mathcal{OC}(e, s) = e \Rightarrow e' \tag{4.2}$$

Enfin,

$$\mathcal{OC}(e, s) = not(e)\ or\ e' \tag{4.3}$$

De même, pour l'opérateur "**OR**" :

$\mathcal{OC}(e, s) = if\ not(e)\ then\ not(e')\ else\ true$

$\mathcal{OC}(e, s) = not(e) \Rightarrow not(e')$

$\mathcal{OC}(e, s) = e\ or\ not(e')$

Pour l'opérateur conditionnel $ite(c, then, else)$, les conditions des chemins unitaires sont triviales. En effet, la sortie s dépend toujours de la valeur de la condition c. En revanche, elle dépend de la valeur de l'entrée e (partie "*then*") si la valeur

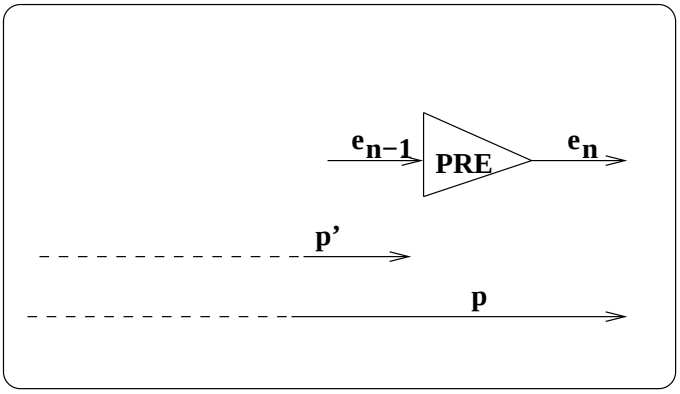

FIG. 4.5 – Construction via l'opérateur de mémoire PRE

de la condition c est *vraie* et dépend de la valeur de l'entrée e' (partie "*else*") si la valeur de la condition c est *fausse*.

2° cas : Opérateur temporel PRE

Dans le cas d'un opérateur temporel PRE (**Fig.** 4.5) , la condition d'activation du chemin est traduite par le fait que le chemin est dit activé si son préfixe p' est activé au cycle précédent. Dans le cas où le chemin p est un chemin unitaire, la condition d'activation associée est simplifiée comme en 4.4.

$$\mathcal{AC}(p) = false \rightarrow pre(true) \tag{4.4}$$

3° cas : Opérateur d'initialisation FBY

Dans le cas d'un opérateur d'initialisation FBY (**Fig**. 4.6), l'équation 4.5 traduit le fait que le chemin est activé si le préfixe est activé à l'instant initial. De même, l'équation 4.6 s'interprète par le fait que le chemin est activé si son préfixe est activé toujours sauf à l'instant initial. Ces deux conditions sont simplifiées, dans le cas de chemins unitaires pour donner les deux équations 4.7 et 4.8 respectivement. La première expression vaut $true$ à l'instant initial et $false$ partout ailleurs. Elle signifie que le chemin p ne peut être activé qu'à l'instant initial ($t = 0$). La seconde décrit

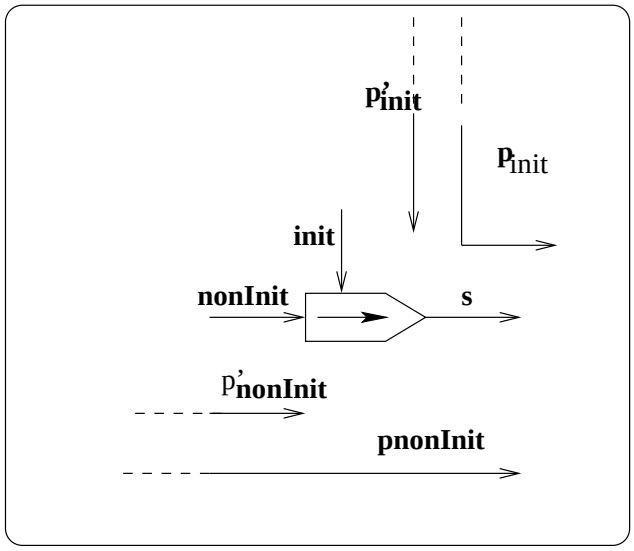

FIG. 4.6 – Condition d'activation de l'opérateur **fby**

un flot de données qui vaut toujours *true* sauf à l'instant initial. Elle signifie que le chemin p ne peut être activé qu'à l'instant t tel que $(t > 0)$.

$$\mathcal{AC}(p) = \mathcal{AC}(p') \rightarrow false \tag{4.5}$$

$$\mathcal{AC}(p) = false \rightarrow AC(p') \tag{4.6}$$

$$\mathcal{AC}(p) = true \rightarrow false \tag{4.7}$$

$$\mathcal{AC}(p) = false \rightarrow true \tag{4.8}$$

Sur le schéma de la figure 4.6, les conditions d'activation associées aux deux chemins p_{init} et $p_{nonInit}$ sont données respectivement par les deux formules 4.9 et 4.10.

$$\mathcal{AC}(p_{init}) = \mathcal{AC}(p'_{init}) \rightarrow false \tag{4.9}$$

$$\mathcal{AC}(p_{nonInit}) = \quad false \rightarrow AC(p'_{nonInit}) \tag{4.10}$$

Exemple 4.5.1 Soit le chemin $p = (Marche, L7, L6, L8, L7, Arrete)$. Pour calculer sa condition d'activation, nous appliquons les règles définies ci-dessus de manière progressive. En partant du dernier arc du chemin et nous le remontons jusqu'au premier arc. Pour cela, nous introduisons au fur et à mesure des chemins intermédiaires pour illustrer la construction.

D'après la règle 4.6, nous avons $AC(p) = false \rightarrow AC(p_1)$ où

$p_1 = (Marche, L7, L6, L8, L7)$.

De même, $AC(p_1) = not(marche)\ and\ AC(p_2)$ où $p_2 = (Marche, L7, L6, L8)$.

Ensuite, $AC(p_2) = false \rightarrow AC(p_3)$ où $p_3 = (Marche, L7, L6)$.

De même, $AC(p_3) = not(marche)\ and\ AC(p_4)$ où $p_4 = (Marche, L7)$.

Et enfin, $AC(p_4) = true$

Par substitution, on obtient l'expression booléenne temporelle suivante qui donne la condition d'activation du chemin p.

$$AC(p) = false \rightarrow (not(marche)\ and\ (false \rightarrow pre(not(marche))))$$

4.5.2 Conditions d'activation et dépliage des opérateurs dans GATEL

En faisant abstraction des deux modèles (réseau d'opérateurs dans notre cas et arbre de test pour GATEL), on constate quelques similitudes entre les notions que nous avons introduites : condition de chemin unitaire et condition d'activation, et celle de prédicat de chemin de l'arbre de test construit par GATEL lors du dépliage des opérateurs (voir section 3.7.5). En effet, le dépliage de l'opérateur $if-then-else$, par exemple, consiste à considérer les deux sous cas où la condition est vraie et où la condition est fausse ; ce que nous exprimons par le biais de deux conditions de chemins unitaires (une condition pour chaque cas).

De même, nos conditions de chemins unitaires associées aux opérateurs booléens

not, *and* et *or* correspondent à des valuations possibles des opérateurs selon les dépliages proposés par GATEL. Par exemple, soit le noeud simple suivant définissant l'opérateur *and*.

```
node pand(A,B :bool) returns (S :bool)
let
   S = A and B ;
tel ;
```

GATEL permet de déplier cet opérateur selon :

```
not A,
not B,
A and B
```

Les conditions des chemins unitaires de (A, S) et (B, S) (voir définition ci-dessus) sont :

```
not A or B
not B or A
```

Les conditions d'activation du critère de couverture des conditions élémentaires pour ces deux chemins mènent aux quatre cas suivants,

```
(not A or B) and A = B and A
(not A or B) and not A = not A
(not B or A) and not B = not B
(not B or A) and B = A and B
```

dont trois sont distincts et correspondent au dépliage de GATEL.

Pour l'opérateur temporel fby, nos conditions d'activation des deux chemins peuvent être assimilées aux notions de statut du cycle (terme employé dans GATEL). L'un des chemins est toujours *activé* au cycle initial et l'autre n'est jamais *activé* au cycle initial mais *activé* partout ailleurs.

Pour l'opérateur temporel *pre*, GATEL n'a pas de "dépliage" explicite mais associe à chaque variable précédée d'un *pre* son évaluation au cycle précédent. Notre

condition d'activation associée à l'opérateur *pre* exprime le fait que le chemin est actif au cycle précédent.

4.5.3 Satisfaction de la condition d'activation

Soit $(i_1, ..., i_m)$ un vecteur de m entrées booléennes ou entières et soient v_l^j $(l = 1...m, j = 1...k)$ k valeurs possible pour chaque i_l.

La condition d'activation du chemin $p = (e_1, ..., e_n)$ est satisfaite si et seulement s'il existe une suite de k affectations $\left(i_1 = v_1^j, ..., i_m = v_m^j\right)$ $(j = 1...k)$ telle que $AC(p) = (false, false, ..., false, true)$.

On dit que la séquence d'entrées $(v_1^1, ..., v_m^1)$,...,$(v_1^k, ..., v_m^k)$ satisfait $AC(p)$ et on note $\mathbb{S}(AC(p), s) = true$.

Exemple 4.5.2 Dans l'exemple du climatiseur (cf. figure 4.3), la séquence $\{(1,0,0,0)$, $(1,1,0,0)$, $(1,1,0,0)\}$ associée au vecteur (*Marche,TempInf,TempOk,TempSup*) est une séquence d'entrées qui permet d'activer le chemin$(Marche, L_7, L_5, Froid)$.

4.6 Critères de couverture : Principes de définition

Les critères dits structurels sont définis sur des éléments sélectionnés dans un modèle de couverture structurel. Ces éléments doivent être facilement repérables. Dans un graphe de flot de contrôle, il s'agit de noeuds et d'arcs représentant respectivement de branchements et des blocs d'instructions. Dans un automate, il s'agit d'états ou de transitions. Dans les deux cas, des éléments composés peuvent être concernés par la couverture : il s'agit des chemins dans les graphes de flot de contrôle et de séquences de transitions dans les automates.

Les premiers travaux sur la couverture des réseaux d'opérateurs [44,46] ont donné lieu à la définition de critères pour la couverture des éléments de base du réseau d'opérateurs, à savoir la couverture de tous les arcs (données) et la couverture de tous les opérateurs (fonctions). Basés sur la notion de prédicat d'arc, ces deux critères fournissent deux moyens simples de mesure de la couverture du réseau d'opérateurs.

Un troisième critère plus fort, la couverture de tous les chemins du réseau, a été proposé. Dans la plupart des cas, ce critère est difficilement satisfiable car le nombre de chemins est souvent infini à cause de la présence des opérateurs temporels.

Un bon critère de couverture est un critère qui offre un bon compromis entre l'efficacité de détection des défauts et le coût de test. Dans les programmes impératifs, un tel compromis est recherché à travers les critères basés sur les métriques TER_i. Ces critères offrent des paliers mesurables et progressifs pour mesurer la couverture d'un graphe de flot de contrôle. Comme la couverture de tous les chemins (100%) est impossible dans certains programmes, à cause de leur nombre potentiellement infini, l'idée étant de couvrir des éléments du graphe dont le nombre est toujours fini. Ces éléments sont soit des instructions, des conditions, des séquences de code sans branchements,...etc. La couverture de toutes les instructions (TER_1), la couverture de toutes les branches (TER_2) et la couverture de tous les LCSAJ (TER_3) sont des exemples de critères permettant de mettre en oeuvre une telle approche.

De manière similaire, la couverture de tous les chemins est impossible à atteindre dans les réseaux d'opérateurs comprenant des chemins avec mémoire. L'idée, donc, est d'introduire une notion de couverture progressive basée sur des sous-chemins du réseau d'opérateurs dont la longueur est toujours finie. Nous proposons, donc, une hiérarchie de critères de couverture applicables aux réseaux d'opérateurs. Cette hiérarchie est basée d'une part sur une construction incrémentale des chemins selon leur longueur et d'autre part sur un raffinement de chaque critère selon sa force.

4.6.1 Couverture de chemins

Dans les programmes impératifs, la couverture des éléments du graphe de flots de contrôle (instructions, branches, chemins,...) est basée sur la notion de chemin d'exécution. Un chemin d'exécution est un chemin reliant le noeud d'entrée du graphe à l'un des noeuds de sortie. Il dénote une exécution possible du programme. Ce chemin est unique pour une donnée d'entrée. Dans les programmes LUSTRE en revanche, la notion de chemin d'exécution telle qu'elle est définie sur les graphes de flot de contrôle n'existe pas sur le réseau d'opérateurs. En effet, l'exécution du programme

sur une donnée d'entrée entraîne l'activation de plusieurs chemins en même temps.

Un programme LUSTRE réalise une fonction associant à une séquence d'entrées une séquence de sortie. Il s'agit en réalité d'un vecteur de fonctions dont chacune calcule la valeur d'une seule variable de sortie et exprime ainsi l'ensemble de dépendances entre cette variable de sortie et les variables d'entrée. Ce sont ces dépendances que nous essayons de mettre en évidence en définissant des chemins dans le réseau d'opérateurs. Elles peuvent être assimilées à des fonctions partielles.

Au lieu de couvrir des éléments de base (arcs et opérateurs) qui composent le réseau d'opérateurs comme le font les critères définis dans [46], les critères que nous proposons sont définis sur des chemins qui sont de longueur quelconque mais finie. La définition de critères de sélection de données de test en termes de couverture de certains chemins du réseau d'opérateurs s'identifie à celle des fonctions significatives parmi les fonctions réalisées par les chemins.

A la différence des critères basés sur le flot de contrôle comme les LCSAJ, la couverture d'un chemin dans un réseau d'opérateurs n'est pas exclusive avec la couverture d'autres chemins. En effet et comme un chemin est une fonction partielle, sa couverture entraîne souvent la couverture d'autres chemins faisant partie de la même fonction. En exigeant la couverture de tous les sous-chemins de longueur différente, les critères fournissent un moyen rudimentaire de forcer le test de toutes les fonctions partielles réalisées par les chemins (ou au moins de tenter de le faire).

Ce type de test est proche aux techniques appliquées au test des circuits digitaux. Cependant, les fautes recherchées dans ce dernier cas sont clairement identifiées (collages).

4.6.2 Construction progressive selon la longueur des chemins

Dans un réseau d'opérateurs, la couverture de tous les chemins est souvent impossible car leur nombre est potentiellement infini, notamment à cause des cycles. En limitant le nombre de cycles, on obtient des chemins de longueur finie, et par conséquent un nombre fini de chemins. Un chemin de longueur n est obtenu par la concaténation d'un chemin de longueur $(n-1)$ et d'un chemin unitaire (longueur

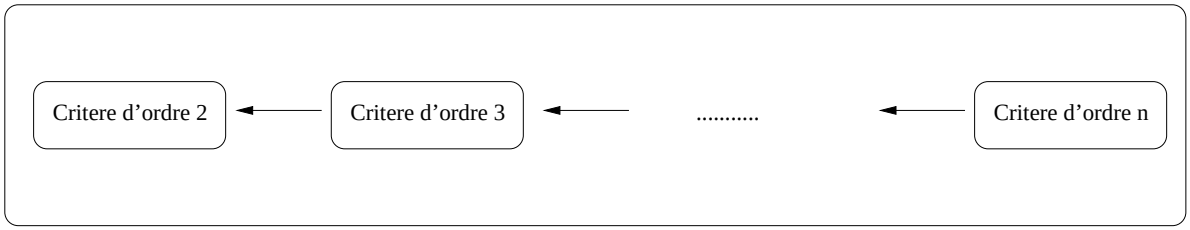

FIG. 4.7 – Inclusion des critères pour des ordres croissants

2).

En partant de chemins unitaires (de longueur 2), des chemins de plus en plus longs sont construits. La construction est finie si le chemin est sans cycles ou si le nombre de cycles est limité. Pour nos critères, nous ne considérons que des chemins de longueur finie.

Définition 4.6.1 *Classe de chemins d'ordre n*

Soit n un entier tel que $n > 1$*, On appelle* classe de chemins d'ordre *n l'ensemble de tous les chemins dont la longueur est inférieure ou égale à n. On note cet ensemble* $P_n = \{p \mid length(p) \leq n\}$

On a :

$$P_2 \subseteq P_3 \subseteq ..... \subseteq P_n$$

A chaque classe de chemins d'ordre i, nous faisons associer un critère. Il est clair que les critères obtenus sont liés par une relation d'inclusion. Cette relation offre un moyen d'établir une forme d'hiérarchie horizontale entre les critères proposés (cf section 4.7)

4.6.3 Raffinement des critères selon leur force

La seconde orientation suggère de définir plusieurs niveaux de force d'un critère d'un ordre quelconque. L'idée est de partir d'un critère de base satisfaisant des contraintes minimales et de le renforcer en rajoutant progressivement des contraintes supplémentaires de manière à rendre plus difficile sa satisfaction.

Pour le critère de base, les contraintes minimales portent sur les conditions d'activation des chemins qu'un jeu de tests doit satisfaire. Selon ce critère, un chemin est dit couvert s'il y a au moins une séquence d'entrées qui satisfait ses contraintes. Intuitivement, ce critère permet de propager la valeur du premier arc du chemin à travers le chemin, et ce sans faire d'hypothèses sur cette valeur. Donc, la forme de dépendance qu'une condition d'activation définit ne se préoccupe pas de la valeur de l'entrée du chemin.

Dans le but d'obtenir une forme de dépendance plus significative et par conséquent un critère plus fort, l'idée est de propager toutes les valeurs possibles de l'entrée d'un chemin booléen. Ceci consiste, donc, à imposer la satisfaction de la condition d'activation du chemin pour toutes les valeurs possibles de l'entrée du chemin (*true* et $false$). Ainsi, si une seule séquence d'entrée suffit à satisfaire le critère de base, deux séquences sont, au minimum, nécessaires pour satisfaire ce second critère. Ce critère est appelé *couverture des conditions élémentaires*

Un troisième critère est obtenu en généralisant la variation effectuée sur l'entrée du chemin à tous les arcs internes booléens du chemin. Ceci consiste à imposer la satisfaction de la condition d'activation du chemin pour toutes les valeurs possibles de chacun des arcs internes du chemin (*true* et $false$). Le nombre de séquences d'entrées nécessaires à satisfaire ce critère augmente en fonction de la longueur du chemin. Ce troisième critère est appelé *couverture des conditions multiples.*

4.7 Critères de couverture structurelle

Dans la suite de ce chapitre, nous considérons un programme LUSTRE dont $\mathcal{N}$ est le réseau d'opérateurs associé et tel que $\mathcal{N}$ contient au moins un opérateur (il existe au moins un chemin unitaire dans le réseau). En plus, les chemins que nous considérons dans la suite de ce chapitre sont tous des chemins complets, i.e, des chemins reliant un arc d'entrée (variable d'entrée du programme) à un arc de sortie (variable de sortie du programme).

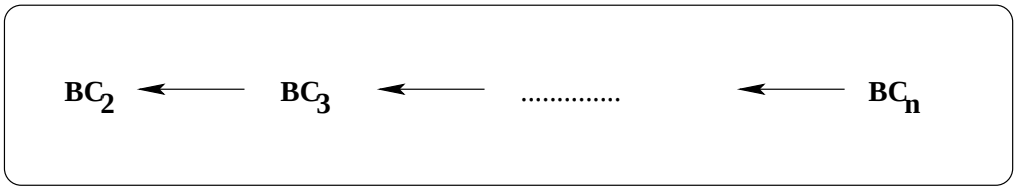

FIG. 4.8 – Inclusion des critères de base

4.7.1 Couverture de base d'ordre n (BC_n)

Ce critère est défini sur une classe de chemins d'un ordre n quelconque ($n \geq 1$ et n est fini). Il est basé sur une notion de couverture simple et intuitive. Selon cette notion, *"couvrir"* un chemin signifie qu'il existe au moins une séquence d'entrées qui permet de l'activer. La satisfaction du critère nécessite d'activer chaque chemin de cette classe au moins une fois. Un chemin p est activé s'il existe une séquence d'entrées qui satisfait sa condition d'activation.

Définition 4.7.1 *Couverture de base d'ordre n*

Soit $\mathcal{P}_n$ l'ensemble de tous les chemins de longueur inférieure ou égale à n , et soit $\mathcal{T}$ un ensemble de séquences d'entrées. On dit que $\mathcal{N}$ est couvert selon la couverture de base d'ordre n si et seulement si :

$\forall p \in \mathcal{P}_n, \exists t \in \mathcal{T} : \mathbb{S}(AC(p), t) = true.$

D'après cette définition, un jeu de tests qui satisfait le critère de base d'ordre n satisfait nécessairement tout critère de base d'ordre k tel que ($k \leq n$). Ceci introduit une relation d'inclusion entre les critères d'ordres croissants.

Remarque:

Dans ce critère, aucune hypothèse n'est faite sur la manière dont l'ordre n est fixé. Le choix de n dépend des objectifs des testeurs et de la complexité du réseau.

La limitation de la longueur des chemins a une interprétation fonctionnelle à savoir que ce nombre correspond à la longueur minimale d'une séquence d'entrées satisfaisant la couverture du chemin.

	Marche	TempInf	TempOk	TempSup
0	*false*	*true*	*true*	*false*
1	*true*	*false*	*false*	*true*
2	*false*	*true*	*false*	*true*

TAB. 4.1 – Séquence satisfaisant le critère BC7

Exemple 4.7.1 La couverture de base d'ordre 7 du réseau d'opérateurs associé au climatiseur (Fig. **??**) est assurée par la séquence d'entrées de longueur 3 illustrée dans la table 4.1.

4.7.2 Couverture des conditions élémentaires d'ordre n (ECC_n)

Motivations

La satisfaction du critère de couverture précédent garantit que tous les chemins d'une certaine classe sont activés. Cette activation ne fait aucune hypothèse sur les valeurs des entrées des chemins. Autrement dit, aucune partition du domaine d'entrée n'est exigée et toute valeur satisfaisant la condition d'activation est considérée comme un cas de test réussi.

Dans ce critère, il suffit de sélectionner une seule valeur sur le domaine de définition de l'entrée qui assure l'activation du chemin. Pour une entrée booléenne, par exemple, il suffit que le chemin soit couvert avec l'une des valeurs *true* (ou $false$) et pas nécessairement les deux. Pour une entrée entière, il suffit de choisir une valeur dans le domaine de définition pour satisfaire le critère.

Considérons l'opérateur *ite* qui possède trois entrées e_1, e_2 et c (condition) et une sortie s. Afin de faire passer les deux arcs e_1 et e_2 dans s, la condition c doit porter une fois la valeur *true* et une fois la valeur $false$. Autrement dit, le chemin unitaire (c, s) doit être couvert une fois avec la valeur *true* et une fois avec la valeur $false$.

La condition d'activation permet de faire passer l'une des valeurs d'entrées (quelle qu'elle soit). Pour la satisfaire, il suffit de trouver une séquence d'entrées qui fait

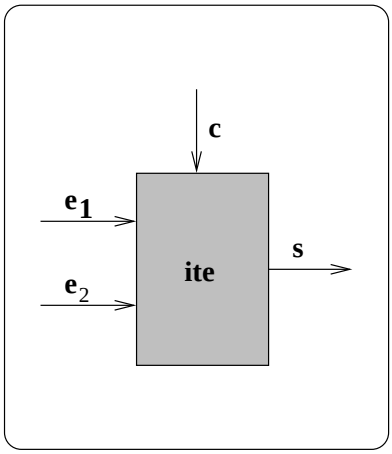

FIG. 4.9 – Opérateur ite

passer $true$ ou $false$ mais pas nécessairement les deux. En effet, lorsque la condition $c = true$ est vérifiée, la sortie s porte la valeur de e_1. La couverture de base de (e, s) est assurée quelque soit la valeur de e_1. Cette valeur peut être $true$ ou $false$ si e_1 est booléen.

Afin de permettre la prise en compte de toutes les valeurs pertinentes des entrées, l'idée est de renforcer la condition d'activation en tenant compte des valeurs des entrées. Ceci permet de mieux mesurer l'impact de la variation des entrées sur la valeur de la sortie.

Pour cela, nous introduisons un nouveau critère qui s'applique à des chemins booléens (chemins dont l'entrée est booléenne). Dans ce critère, un chemin est couvert si sa condition d'activation est satisfaite avec chacune des valeurs de son entrée.

Définition 4.7.2 *Couverture des conditions élémentaires - ECC*

Soit $\mathcal{P}_n$ l'ensemble de tous les chemins de longueur inférieure ou égale à n , et soit $\mathcal{T}$ un ensemble de séquences d'entrées. On dit que $\mathcal{N}$ est couvert selon la couverture des conditions élémentaires d'ordre n si et seulement si : $\forall p \in \mathcal{P}_n$

$$\exists t_1 \in \mathcal{T},\ \mathbb{S}(AC(p), t_1) = true. \ \wedge\ first(p).value = true$$

$$\exists t_2 \in \mathcal{T} :\ S(AC(p), t_2) = true \ \wedge\ first(p).value = false$$

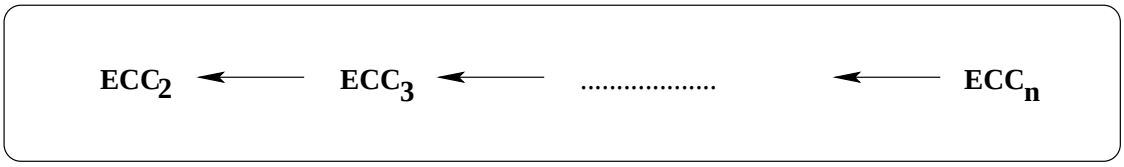

FIG. 4.10 – Inclusion des critères de couverture des conditions élémentaires

	Marche	TempInf	TempOk	TempSup
0	*false*	*true*	true	*false*
1	*true*	*false*	*false*	*true*
2	*false*	*true*	*false*	*true*
3	*true*	*true*	*true*	*false*
4	*true*	*true*	*false*	*true*
5	*true*	*true*	*true*	*true*
6	*false*	*false*	*true*	*true*
7	*false*	*false*	*true*	*true*
8	*true*	*false*	*true*	*false*

TAB. 4.2 – Séquence satisfaisant le critère ECC7

Il est clair que le critère de couverture d'ordre 3 inclut le critère d'ordre 2, et que le critère d'ordre 4 inclut ceux d'ordre 2 et 3...et ainsi de suite. De façon plus générale, le critère d'ordre $(n+1)$ inclut toujours le critère d'ordre n et ce pour tout $n \geq 2$ (cf. section 4.10).

La couverture des conditions élémentaires d'ordre 7 du réseau d'opérateurs du climatiseur est assurée par la séquence d'entrées de longueur 9 illustrée dans la table 4.2.

Complexité du critère

Si le nombre de chemins $\mathcal{P}_n$ est donné par M alors le nombre de conditions à satisfaire pour le critère de conditions élémentaires est de $2 * M$.

Par rapport au critère de couverture de base, la difficulté de satisfaire le critère de conditions élémentaires réside dans la variation effectuée au niveau de entrées des chemins.

Soit un entier $k \leq n$ et soit m_k le nombre de chemins de longueur k. Le nombre

de conditions d'activation associées au critère ECC d'ordre n est $M = 2 * (m_2 + m_3 + ... + m_n)$

$$M = 2 * \sum_{k=2}^{k=n} m_k$$

Contrairement au critère de couverture de base, la satisfaction de ce critère n'est pas toujours garantie par un jeu des tests, non pas parce que le jeu de tests est insuffisant mais à cause de certains chemins qui ne sont jamais activés avec les deux valeurs $vrai$ et $faux$.

Critères proches ou similaires

Le critère de couverture des conditions élémentaires (ECC) est similaire à la couverture des décisions DC (Decision Coverage) pour les expressions booléennes. En effet, le critère DC (Couverture de décisions) nécessite que chaque décision soit exécutée au moins une fois avec la valeur $true$ et une fois avec la valeur $false$. Les mêmes similitudes peuvent être constatées avec la couverture des branches dans les graphes de flot de contrôle [63] et le critère "couverture des branches" appliqué aux graphes causes-effets [45, 53].

4.7.3 Couverture des conditions multiples d'ordre n (MCC_n)

Motivations

Dans les modèles à base de flots de données, l'analyse d'un chemin fait ressortir les dépendances entre la sortie et l'entrée du chemin. Autrement dit, elle examine l'impact de l'entrée sur la sortie. Cependant, la valeur de la sortie ne dépend pas uniquement de la valeur de l'entrée du chemin mais dépend également de tous les arcs internes. Dans le critère de couverture des conditions élémentaires, seule la dépendance entre l'entrée du chemin et sa sortie est prise en compte. Ce critère néglige toutes les autres dépendances de la sortie avec les arcs locaux.

Soit le réseau d'opérateurs de la figure 4.11 et soit le chemin booléen $p = (a, L_1, s, o)$. La couverture selon le critère des conditions élémentaires de ce che-

```
node extension(a,b,c,d : bool) returns (o : bool)
var s : bool ;
let
 s = (a and b) or c ;
 o = if s then (a or b) else (not c) ;
tel ;
```

FIG. 4.11 – Exemple d'une expression incluant un opérateur if-then-else

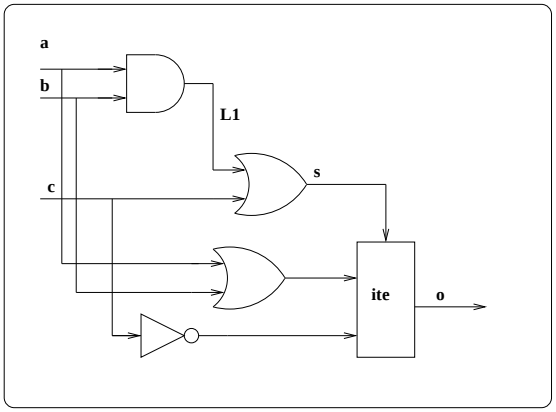

FIG. 4.12 – Réseau d'opérateurs du noeud "extension"

min nécessite de satisfaire sa condition d'activation avec les deux valeurs possibles de son entrée a. Ceci permet de mesurer l'impact de la variable a sur la sortie o sans imposer une variation au niveau des arcs locaux L_1 et s ; or ce dernier est "*l'entrée-condition*" d'un opérateur *ite* dont la valeur permet de faire passer soit la valeur de l'arc de la partie *then* soit la valeur de l'arc de la partie *else*.

Afin de permettre de faire passer à la fois les valeurs des deux arcs associés à la partie *then* et à la partie *else*, il est nécessaire d'effectuer une variation au niveau de l'arc s. En d'autres termes, ceci nécessite d'affiner la couverture du chemin p en donnant à chacun des arcs internes une fois la valeur *true* et une fois la valeur $false$.

De manière générale, ceci revient à explorer toutes les combinaisons des arcs (y compris les arcs locaux). ces nouvelles exigences donnent lieu à un nouveau critère de couverture plus fort que le critère des conditions élémentaires. Selon ce critère, un chemin est couvert si sa condition d'activation est satisfaite avec chacune des valeurs possibles des arcs qui le composent. Ce critère est formalisé comme suit :

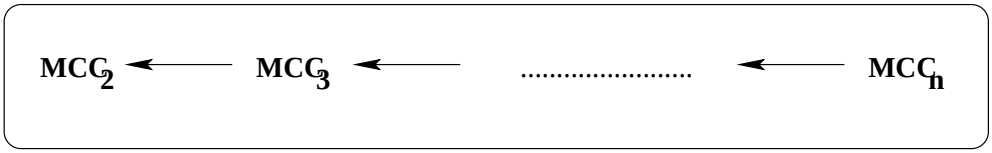

FIG. 4.13 – Inclusion des critères de couverture des conditions multiples

Définition du critère

Définition 4.7.3 *Couverture des conditions multiples – MCC*

Soit $\mathcal{P}_n$ l'ensemble de tous les chemins de longueur inférieure ou égale à n , et soit $\mathcal{T}$ un ensemble de séquences d'entrées. On dit que $\mathcal{N}$ est couvert selon la couverture des conditions multiples d'ordre n si et seulement si : $\forall p \in \mathcal{P}_n, \forall e \in p$

$$\exists t_1 \in \mathcal{T},\ \mathbb{S}(AC(p), t_1) = true. \ \wedge \ e.value = true$$

$$\exists t_2 \in \mathcal{T} :\ S(AC(p), t_2) = true \ \wedge \ e.value = false$$

Les critères $MCC_i (i = 2, 3, ...n)$ sont comparables entre eux. En effet, la satisfaction du critère d'ordre n garantit nécessairement la satisfaction de tous les critères d'ordres inférieurs.

Complexité du critère

Pour le critère de couverture de conditions multiples, le nombre de combinaisons croît de façon exponentielle avec l'ordre. En effet, la couverture d'un chemin de longueur k nécessite la satisfaction de $(2 * (k-1))$ conditions d'activation. Si m_k dénote le nombre de chemins de longueur $k \leq n$ (m_2 chemins de longueur 2, m_3 chemins de longueur 3, ..., m_n chemins de longueur n) alors le nombre de conditions d'activation d'ordre n est donné par M tel que :

$M = 2 * (m_2 + 2 * m_3 + ... + (n-1) * m_n)$

$M = 2 * \sum_{k=2}^{k=n} ((k-1) * m_k)$

Critères proches ou similaires

Pour des chemins où la condition d'activation est une simple expression booléenne (sans opérateurs temporels), le critère de couverture des conditions multiples a les mêmes exigences que le critère de couverture des conditions multiples (FPC) [43]. Ce critère exige que toutes les conditions atomiques dans une condition composée soient exécutées avec les deux valeurs possibles ($false$ et $true$) ainsi que la condition composée elle-même.

4.7.4 Relations d'inclusion

Il existe deux formes d'inclusion entre les critères définis ci-dessus : une inclusion intra-critères, qui concerne les critères de la même classe de chemins (BC, ECC, MCC) et une inclusion inter-critères qui concerne une comparaison des critères entre eux.

Inclusion intra-critères

Les trois relations d'inclusion ci-dessous sont résumées par les flèches horizontales de la figure 4.14. Le facteur discriminant est la longueur des chemins. Cette forme d'inclusion entre critères de même famille est d'ordre ensembliste. En effet, la classe de chemins d'ordre n ($\mathcal{P}_n$) est définie comme étant la classe de chemins d'ordre $n-1$ ($\mathcal{P}_{n-1}$) à laquelle s'ajoute l'ensemble des chemins du réseau dont la longueur est égale à n.

$P_n = \{p \mid length(p) = n\}$

$\mathcal{P}_n = \mathcal{P}_{n-1} \cup\ P_n$

Par conséquent, l'ensemble de conditions d'activation d'une classe d'ordre $(n-1)$ est un sous-ensemble de l'ensemble des conditions d'activation de la classe d'ordre n. Il est évident, donc, que toute séquence d'entrées satisfaisant les conditions d'activation d'un critère d'ordre n satisfait nécessairement les conditions d'activation d'un critère d'ordre $(n-1)$.

Inclusion inter-critères

L'analyse des conditions sous-jacents aux critères fait ressortir les formes d'inclusion illustrées par les flèches verticales de la figure 4.14. En effet,

- La condition pour satisfaire un critère de condition élémentaire d'ordre quelconque est plus forte que la condition à satisfaire pour le critère de base du même ordre. Pour un chemin p, la condition à satisfaire pour le critère de base est la condition d'activation $\mathcal{AC}(p)$ tandis que pour le critère de conditions élémentaires, il faudrait que cette condition soit satisfaite pour les deux valeurs possibles de l'entrée du chemin : $vrai$ ($first(p) \wedge \mathcal{AC}(p)$ et $faux$ ($not(first(p)) \wedge \mathcal{AC}(p)$. Ce qui est plus fort que la condition de condition elle-même.
- La condition pour satisfaire un critère de conditions multiples d'ordre quelconque est plus forte que la condition à satisfaire pour le critère de conditions élémentaires du même ordre. Pour un chemin p, la satisfaction du critère de conditions élémentaires nécessite deux conditions $first(p) \wedge \mathcal{AC}(p)$ et $not(first(p)) \wedge \mathcal{AC}(p)$ tandis que pour le critère de conditions multiples, il faudrait autant de conditions sur chacun des arcs intermédiaires ce qui contraint davantage la couverture du chemin.

Cependant, un critère BC d'ordre n n'est pas comparable avec les critères ECC et MCC d'ordres inférieurs. De même que le critère ECC d'ordre n quelconque est incomparable avec aucun critère MCC d'ordre inférieur.

Relation avec les critères def-use

Bien que les critères que nous avons proposés ont les mêmes motivations que les critères de couverture basés sur l'analyse des flux de données définis sur des graphes de flot de contrôle, il n'existe aucune relation d'inclusion simple entre ces critères et les critères définis plus haut. En effet, il est difficile de faire la distinction entre une définition et une utilisation d'une variable dans un programme constituée

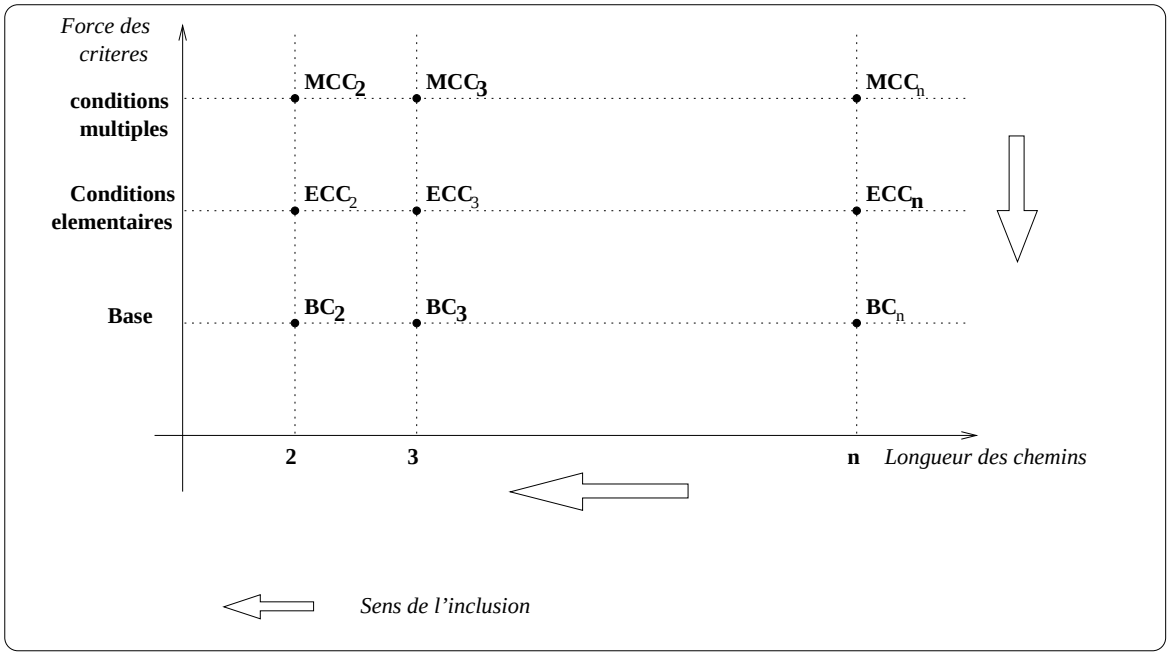

FIG. 4.14 – Hiérarchie des critères

d'équations au sens mathématique où la partie droite de l'équation est la même que la partie gauche.

4.8 Méthodologie de couverture à base des critères

Comme le montre l'algorithme ci-dessus, la mesure de la couverture est effectuée en appliquant les critères de manière progressive. La progression est réalisée à deux niveaux :

1. Une progression intra-critère, qui consiste en une variation croissante de l'ordre (longueur de chemins). Un ordre minimal est fixé en fonction de la complexité du réseau d'opérateurs. Ensuite, cet ordre est augmenté jusqu'à une certaine limite qui peut dépendre du nombre de cycles dans le réseau.
2. Une progression inter-critères consistant à passer d'un critère faible (couverture de base) aux critères plus fort (couverture des conditions élémentaires ensuite couverture des conditions multiples). Cette variation est appliquée lorsque une satisfaction des critères de base est obtenue.

Algorithm 2 Mesure de la couverture avec LUSTRUCTU

```
Début
     Critère_Choisi = BC ;
 L1 : Fixer n ;
 L2 : Appliquer Critère_Choisi(n)
     Si (ratio_couverture = 100) alors
      Augmenter (n) ;
      Aller à L2 ;
     Si (n = Max)
      Critère_Choisi = ECC ;
      Aller à L1 ;
     Si (n = Max)
      Critère_Choisi = MCC ;
      Aller à L1 ;
Fin ;
```

4.9 Discussion et comparaison

Nous terminons ce chapitre par une brève comparaison avec les deux types de couverture les plus proches de notre approche à savoir la couverture structurelle dans GATEL et la couverture dans le domaine du test du matériel.

4.9.1 Couverture structurelle dans GATEL

Comme nous l'avons vu au chapitre 3, GATEL est un outil de génération de séquences de tests guidée par un objectif de test. Cette génération est basée sur une résolution d'un système de contraintes dérivé à partir de l'objectif de test, du programme et des contraintes de l'environnement.

L'outil GATEL ne définit pas de critère de mesure de la couverture du programme LUSTRE. Cependant, il offre à l'utilisateur la possibilité d'envisager pour une sortie, pendant la génération, une couverture structurelle des chemins de calcul dans l'arbre de test associé à cette sortie (voir section 3.7.5).

De plus, il n'y a aucun moyen explicite de relier les chemins de calcul dans les arbres de test aux chemins dans le réseau d'opérateurs modélisant le programme. Enfin, notre approche consiste à définir des critères qui sont destinés à mesurer la couverture d'un programme LUSTRE, à posteriori, sans rapport avec un objectif de

```
node srl(es,er :bool)
returns (s :bool)
let
  s = if er then false
      else if es then true
      else false->pre s ;
tel ;
```

FIG. 4.15 – Le noeud set-reset latch

génération.

Par ailleurs, notre approche et la génération guidée par des objectifs de test dans GATEL se complètent dans le sens où nos critères peuvent être facilement utilisés pour guider la génération. L'utilisation conjointe de GATEL avec nos critères a été expérimentée sur une étude de cas du monde de l'avionique [8].

Le premier problème était de caractériser un critère de couverture adapté à LUSTRE en vue de guider la génération de cas de tests par l'outil GATEL. Habituellement, les testeurs identifient des classes d'équivalence fonctionnelle et des points singuliers. Un bon critère permet d'avoir un cas de test par classe d'équivalence. L'idée est que des classes d'équivalence sont définies par des chemins dans le réseau d'opérateurs du programme LUSTRE et que le critère recherché est celui qui couvre ces chemins.

Dans ce travail, nos critères ont été utilisés comme règles renforçant les règles de dépliage implantés par GATEL pour guider le processus de génération, et en particulier le critère de couverture de conditions élémentaires (ECC). A travers ce critère, notre approche apporte un niveau de décomposition (découpage) des opérateurs plus fin que celui opéré actuellement par GATEL.

Soit le noeud LUSTRE définissant un symbole slr (set-reset latch) (voir **Fig.** 4.15).

Afin de définir des règles de dépliage pour ce symbole, nous appliquons les principes de construction des conditions d'activation correspondant au critère de conditions élémentaires.

Nous obtenons les cas suivants :

```
er
```

```
not er ,
es and not er,
not es and not er,
false->(pre s and not er and not es),
false->(not pre s and not er and not es).
```

Le second cas étant couvert par le troisième et le quatrième, il peut être écarté. Le quatrième cas est partiellement couvert par les deux derniers. Afin d'assurer sa couverture à l'état initial, nous rajoutons le cas :

```
(not es and not er)->false
```

Nous obtenons, ainsi, les cinq cas suivants :

```
er
es and not er,
(not es and not er)->false
false->(pre s and not er and not es),
false->(not pre s and not er and not es).
```

4.9.2 Couverture structurelle des circuits digitaux

Notre approche et les approches de test des circuits digitaux s'appuient sur des modèles quasi-similaires. Nous utilisons les réseaux d'opérateurs avec les mêmes notations que celles des circuits digitaux (portes logiques, switches, opérateurs de retard,...etc). De même, les deux approches partagent l'idée de propagation d'un signal à travers les opérateurs (portes).

En termes de critères de couverture, le test des circuits se distingue de notre approche en deux points :

1. La couverture dans les circuits digitaux s'appuie sur des modèles de fautes à base de collage essentiellement. En effet, les fautes dans le domaine du matériel (fautes de fabrication et de conception) peuvent être identifiées et répertoriées ce qui n'est pas le cas pour les programmes LUSTRE pour lesquels il n'y a pas de modèles de fautes reconnus.

2. L'objectif des critères dans le test des circuits digitaux est de générer des vecteurs de test qui assurent un certain niveau de couverture du modèle de fautes tandis que nos critères visent à fournir des mesures qui ne s'appuient sur aucun modèle de fautes et qui permettent entre autres d'évaluer la qualité des tests.

4.10 Conclusion

Dans ce chapitre, nous avons proposé une hiérarchie de critères de couverture adaptés au langage LUSTRE. Utilisant le réseau d'opérateurs comme modèle de couverture, les critères tiennent compte des particularités du langage, à savoir la dimension temporelle et la nature flot de données. La notion de couverture que nous avons introduite est définie sur des classes de chemins du réseau dont la longueur est finie, et repose sur une analyse des dépendances entre les données.

La hiérarchie est constituée de trois critères de force croissante (critère de base, critère de couverture des conditions élémentaires et critère de couverture des conditions multiples). Les critères présentent certaines similitudes avec les critères appliquées aux expressions booléennes soit sur des graphes de flot de contrôle soit sur les graphes causes-effets.

Les définitions formelles présentées dans ce chapitre facilitent l'implémentation des critères. Ces résultats seront validées par des expérimentations montrant l'applicabilité et la pertinence de ces critères dans le chapitre suivant.

Chapitre 5

Réalisations et Expérimentations

"When you can measure what you are speaking about, and express it in numbers, you know something about it, but when you cannot measure it, when you can not express it in numbers, then your knowledge is of a meager and unsatisfactory kind, it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced it to the stage of science"

Lord Kelvin, Physicien Anglais, 1882

5.1 Introduction

A la suite de l'étude théorique qui a donné lieu à la définition formelle des critères présentés au chapitre précédent, nous avons réalisé un ensemble de développements et d'expériences pratiques que nous résumons dans la suite de ce chapitre.

Dans un premier temps, nous présentons LUSTRUCTU, un outil à l'état de prototype, que nous avons réalisé dans le but de valoriser le travail de formalisation des critères, de montrer leur pertinence et d'automatiser le processus de mesure via ces critères. Pour cela, nous décrivons, ensuite, une étude de cas extraite d'une application du domaine de l'avionique sur laquelle nous avons effectué un certain nombre de mesures. La taille et la complexité de l'application sont telles que les résultats obtenus fournissent une première validation de la pertinence des critères définis.

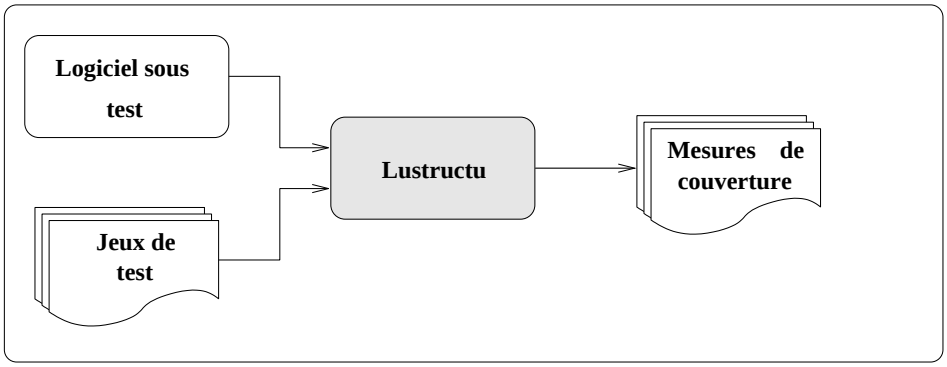

FIG. 5.1 – LUSTRUCTU : Outil de mesure de la couverture

5.2 LUSTRUCTU

LUSTRUCTU (LUstre STRUCTUrel) est un outil de mesure de la couverture des programmes LUSTRE. L'outil prend comme entrées le programme LUSTRE sous test, et un jeu de tests, sous forme de séquences d'entrées (voir Fig. 5.1) et fournit des taux de couverture selon des critères qu'il implémente. L'objectif de l'outil est multiple :

1. Implanter les critères de couverture
2. Automatiser le processus de mesure de la couverture en utilisant ces critères
3. Automatiser les tâches liées au processus de test (compilation, sélection des données de test, soumission et exécution).
4. Assister à l'évaluation de la pertinence des critères.

L'architecture de LUSTRUCTU est composée d'un noyau, développé en C++, implantant les trois critères et d'un certain nombre de scripts, écrits en Perl, en amont et en aval pour automatiser le processus de mesure de la couverture.

5.2.1 Éléments d'implémentation

Réseau d'opérateurs

Le réseau d'opérateurs est implanté par une structure arborescente multi-entrées. Basé sur la relation de successeur qui fournit pour chaque arc (variable) un ensemble d'arcs successeurs, un algorithme récursif parcourt le programme sous test et produit une structure dont chaque élément représente un arc et les liens vers ses successeurs.

La profondeur de l'arborescence dépend du nombre de successeurs directs et indirects d'une entrée. Un cycle est présent dans une arborescence si un arc est successeur indirect de lui-même.

La construction de l'arborescence est effectuée de manière dynamique au moment de l'implémentation d'un critère (l'ordre est connu ce qui donne une arborescence dont la profondeur est finie). La présence de cycles dans le graphe se traduit par la duplication de la portion du graphe qui constitue le cycle.

Chemins

Comme nous l'avons vu au chapitre précédent, chaque critère de couverture (BC, ECC ou MCC) est associé à un ordre (longueur maximale des chemins à considérer), dont l'intérêt est de rendre la longueur des chemins à examiner finie. De manière implicite, limiter la longueur d'un chemin permet de limiter le nombre de cycles (s'il y en a) dans ce chemin.

Structurellement, un cycle dans un chemin est défini comme étant la répétition d'une partie du chemin à cause de la présence de l'opérateur temporel "PRE". Fonctionnellement, le nombre de cycles fait référence aux nombres d'instants (tics d'horloge) que comprend le chemin. Un chemin qui comprend n cycles est un chemin dont un ou plusieurs flots (arcs) sont définis sur un passé de n instants. Dans la définition formelle des critères, nous n'avons pas fait référence de manière explicite au nombre de cycles, car le fait de considérer des chemins de longueur finie suppose implicitement que le nombre de cycles potentiels dans le chemins est fini également. Dans l'implémentation, en revanche, nous y faisons référence de façon explicite et ce pour deux raisons :

1. La première, d'ordre théorique, nous permet d'introduire, outre la longueur, un niveau de discrimination supplémentaire entre les chemins. Ceci permet, entre autres, de mesurer la variation de la couverture non seulement en fonction de la longueur mais aussi en fonction du nombre de cycles.
2. La seconde, d'ordre pratique, en nous permettant d'optimiser la construction de l'arborescence. En effet, un nombre de cycles correspond au nombre de

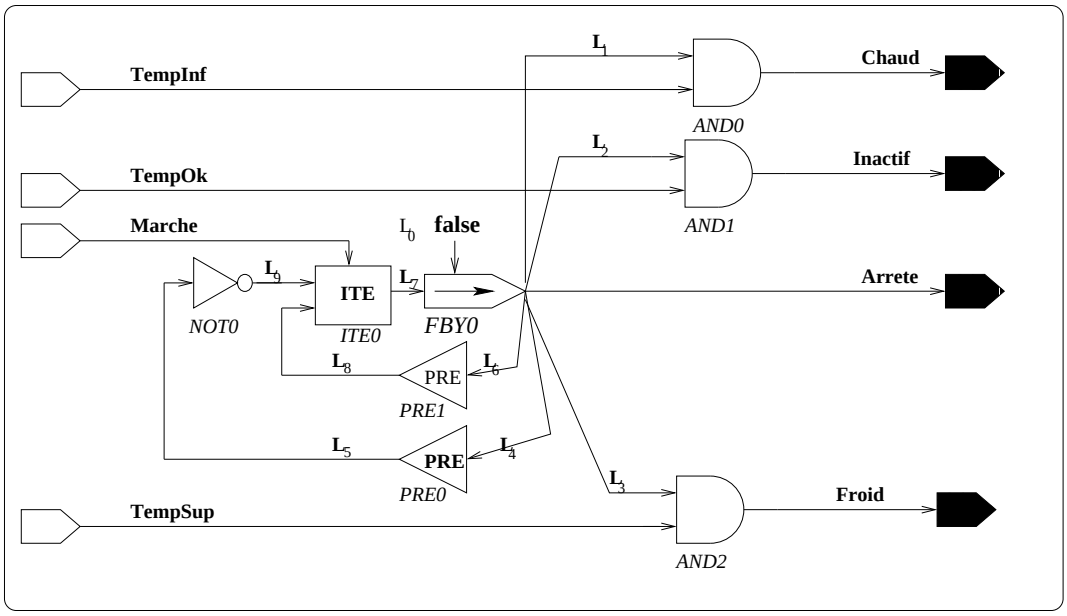

FIG. 5.2 – Réseau d'opérateurs du noeud Climatiseur

fois où la portion qui précède le *pre* est dupliquée. Ce qui représente un gain significatif en termes de représentation du graphe.

Conditions d'activation

Telle que définie dans le chapitre précédent, une condition d'activation est une expression booléenne temporelle associée à chaque chemin du réseau. Dans LUSTRUCTU, le calcul des conditions d'activation s'effectue à la volée lors du calcul des chemins. Selon le type de chemin, on distingue deux types de construction :

- Pour les chemins élémentaires (sans mémoire) où il n'y a pas de cycles, la construction est *itérative* par juxtaposition (conjonction) de conditions des chemins unitaires composant le chemin.
- Pour les chemins avec mémoire où il y a des cycles, la construction est *récursive* par duplication de l'opérateur *pre*. Ce type de conditions d'activation nécessite l'identification des parties des chemins sur lesquelles porte le cycle.

Noeud de couverture

Les conditions d'activation relatives à un critère de couverture d'un ordre quelconque forment un ensemble de contraintes représentant chacune la condition nécessaire pour couvrir un chemin du réseau. Pour un jeu de tests donné, notre objectif consiste à calculer le nombre de chemins couverts, i.e., le nombre de conditions d'activation qui sont satisfaites après exécution sur un jeu de tests donné. Pour cela, l'ensemble de toutes les conditions associées à un critère donné sont regroupé dans un programme LUSTRE nommé *noeud de couverture*.

Associé à un critère, un **noeud de couverture** est un programme (noeud) LUSTRE dont la fonction est de calculer le taux de couverture du programme sous test selon ce critère. La figure 5.3 illustre la structure syntaxique d'un noeud de couverture.

1. Les entrées du noeud de couverture $i_1 : t_1, ..., i_m : t_m$ sont les entrées du programme sous test (variables munies de leurs types respectifs).
2. La sortie est une variable entière (*ratio*) qui retourne le taux de couverture. La variable entière *ratio* est un compteur qui cumule le nombre de conditions d'activation qui ont été vraies au moins une fois pendant l'exécution.
3. Les sorties du programme sous test deviennent des variables locales.
4. L'ensemble des équations du noeud de couverture est formé :
   (a) des équations du programme sous test.
   (b) des équations de la forme $AC_i = f(i_1, i_2, ..., i_m)$ qui définissent les conditions d'activation des chemins.
   (c) des équations de la forme $S_i = AC_i \rightarrow (AC_i \; or \; pre(S_i))$ telle que S_i est une variable booléenne qui calcule la satisfaction de la condition d'activation AC_i du $i - eme$ chemin. Chaque équation vérifie la satisfaction de la condition d'activation (AC_i) du $i - eme$ chemin sur une séquence d'entrées. Notre objectif étant de vérifier que la condition d'activation est vraie après l'exécution de noeud de couverture sur une séquence d'entrées. Si la séquence d'entrées sur laquelle le noeud de couverture est exécuté

```
node criterion_order(i_1 : t_1, ....., i_m : t_m)
returns (ratio : int)
var
  -- variables locales
let
    -- Équations du programme sous test
    -- Équations de calcul des conditions d'activation
    -- Équations de calcul du taux de couverture
tel ;
```

FIG. 5.3 – Syntaxe d'un noeud de couverture

N° chemin	Chemin	Condition d'activation
1	$(Marche, L7, Arrete)$	$AC_1 = false \rightarrow (false \rightarrow true)$
2	$(TempInf, Froid)$	$AC_2 = not(TempInf)$ *or* $Arrete$
3	$(TempOk, Inactif)$	$AC_3 = not(TempOk)$ *or* $Arrete$
4	$(TempSup, Chaud)$	$AC_4 = not(TempSup)$ *or* $Arrete$

TAB. 5.1 – Chemins et Conditions d'activation

est d'une longueur k, il suffit que la condition d'activation soit *vraie* pour une seule entrée pour que l'équation soit vraie.

(d) des équations de la forme $ratio_i = (if\ S_i\ then\ 1\ else\ 0) + ratio_{i-1}$ qui calculent le cumul de conditions d'activation satisfaites par une séquence d'entrées. La variable entière $ratio_i$ incrémente $ratio_{i-1}$ d'une unité si la condition d'activation du $i - eme$ chemin est satisfaite. La variable de sortie correspond donc au dernier $ratio_i$.

Dans l'exemple du climatiseur, le noeud de couverture qui correspond à la couverture de base d'ordre 3 (BC_3) est donné dans la figure 5.4. Comme l'indique l'entête du noeud de couverture, ce programme mesure la couverture du programme sous test selon le critère de couverture de base d'ordre 3. Les entrées du noeud sont ceux du programme sous test (les variables booléennes *Marche, TempInf, TempOk* et *TempSup*) et la sortie est un entier qui donne le score (taux) de couverture (*ratio4*). Les chemins dont la longueur est inférieure ou égale à 3 sont résumés dans la table 5.1.

```
node BC_003(Marche, TempInf, TempOk, TempSup :bool)
returns (ratio4 : int)
var
  L9, L8, L7, L5, L0 : bool;
  Arrete : bool;
  Chaud : bool;
  Inactif : bool;
  Froid : bool;
  AC4, AC3, AC2, AC1  : bool;
  S4, s3, s2, s1 : bool;
  ratio3, ratio2, ratio1 : int;
let
  Arrete = L0 -> L7;
  L5 = pre(Arrete);
  L8 = pre(Arrete);
  L9 = not(L5);
  Froid = Arrete and TempInf;
  Chaud = Arrete and TempSup;
  Inactif = Arrete and TempOk;
  L7 = if Marche then L9 else L8;
  L0 = false;

-- Activation conditions of sub paths
  -- (Marche ,L7 ,Arrete)
  AC1 = false-> (false->true);
  S1 = AC1->(AC1 or pre(S1));
  ratio1 = if S1 then 1 else 0;
  -- (TempInf ,Froid)
  AC2 = not(TempInf) or Arrete;
  S2 = AC1->(AC2 or pre(S2));
  ratio2 = (if S2 then 1 else 0) + ratio1;
  -- (TempOk ,Inactif)
  AC3 = not(TempOk) or Arrete;
  S3 = AC3->(AC3 or pre(S3));
  ratio3 = (if S3 then 1 else 0) + ratio2;
  -- (TempSup ,Chaud)
  AC4 = not(TempSup) or Arrete;
  S4 = AC3->(AC4 or pre(S4));
  ratio4 = (if S4 then 1 else 0) + ratio3;
tel;
```

FIG. 5.4 – Calcul du taux de couverture

```
node Criterion_order_paths(i1 :t,i2 :t,.....)
returns (S1,S2,...,Sn : bool)
 var
 -- variables locales
let
    -- Equations du programme sous test
    -- Equations de calcul des conditions d'activation
tel ;
```

FIG. 5.5 – Noeud calculant les chemins couverts

Chemins non couverts

Le calcul du taux de couverture, via des noeuds de couverture, permet de fournir une mesure quant au pourcentage des chemins qui sont couverts. Lorsque la couverture atteinte est inférieure à 100%, il est utile de savoir lesquels des chemins n'ont pas été couverts et s'il s'agit de chemins infaisables. Pour cela, une variante des noeuds de couverture ci-dessus est construite afin de fournir les chemins qui ne sont jamais couverts pendant le processus de mesure de la couverture. Cette variante est un noeud LUSTRE dont la structure est proche de la structure du noeud de couverture (voir Fig. 5.5). Dans ce noeud, les sorties sont des variables booléennes $S_1, S_2, ..., S_n$ qui décrivent respectivement la satisfaction des conditions d'activation associées à chacun des chemins du réseau.

Afin d'identifier les chemins qui sont couverts et les chemins qui ne sont pas couverts dans l'exemple du climatiseur, le noeud de la figure 5.6 permet de vérifier les conditions d'activation qui sont satisfaites par des séquences d'entrées. Les sorties booléennes S1, S2, S3 et S4 correspondent à la satisfaction des conditions d'activation AC1, AC2 , AC3 et AC4.

5.2.2 Architecture et fonctionnement de LUSTRUCTU

Architecture de LUSTRUCTU

Comme le montre la figure 5.8, LUSTRUCTU [27], à l'état de prototype, est constitué d'un noyau et d'un ensemble de scripts qui permettent d'automatiser la mesure de couverture.

```
node BC_003(Marche, TempInf, TempOk, TempSup :bool)
returns (S1,S2,S3,S4 : bool)
var
  L9, L8, L7, L5, L0 : bool;
  Arrete : bool;
  Chaud : bool;
  Inactif : bool;
  Froid : bool;
  AC4, AC3, AC2, AC1  : bool;
let
  Arrete = L0 -> L7;
  L5 = pre(Arrete);
  L8 = pre(Arrete);
  L9 = not(L5);
  Froid = Arrete and TempInf;
  Chaud = Arrete and TempSup;
  Inactif = Arrete and TempOk;
  L7 = if Marche then L9 else L8;
  L0 = false;

  -- Activation conditions of sub paths
  -- (Marche ,L7 ,Arrete)
  AC1 = false-> (false->true);
  S1 = AC1->(AC1 or pre(S1));
  -- (TempInf ,Froid)
  AC2 = not(TempInf) or Arrete;
  S2 = AC1->(AC2 or pre(S2));
  -- (TempOk ,Inactif)
  AC3 = not(TempOk) or Arrete;
  S3 = AC3->(AC3 or pre(S3));
  -- (TempSup ,Chaud)
  AC4 = not(TempSup) or Arrete;
  S4 = AC3->(AC4 or pre(S4));
tel;
```

FIG. 5.6 – Calcul des chemins couverts

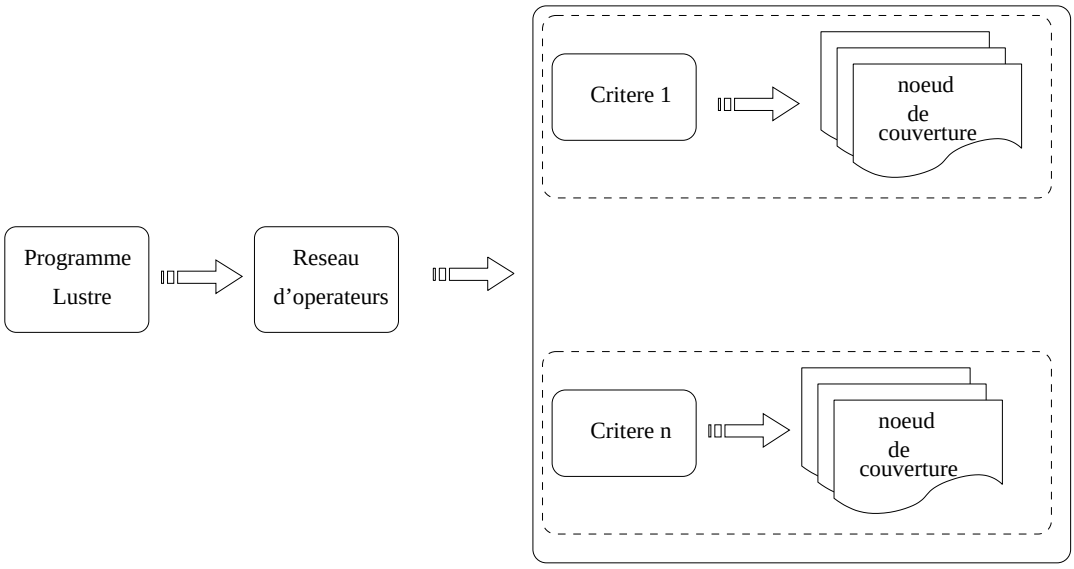

FIG. 5.7 – Principe d'implémentation de LUSTRUCTU

Le **noyau** est le coeur de l'outil. Sa fonction est d'implanter les trois types de critères de couverture. Il se compose de deux modules :

- Un compilateur qui permet de compiler le programme LUSTRE (le code LUSTRE du programme sous test fourni en entrée) pour produire le réseau d'opérateurs associé.
- Un générateur pour engendrer des noeuds de couverture (et les binaires associés) qui implémentent les conditions d'activation relatives à chaque critère de couverture pour produire des noeuds de couverture. Chaque noeud de couverture est identifié par le critère qu'il met en oeuvre (critère de base, critère de couverture de conditions élémentaires, critère de couverture de conditions multiples) et son ordre (2, 3,....n).

Outre le noyau qui permet de construire le réseau d'opérateurs et engendrer les noeuds de couverture, l'outil est doté d'un certain nombre de scripts dont le rôle est de rendre le processus de mesure de la couverture complètement automatique (sans intervention du testeur). Ces scripts sont :

1. *Un script de compilation* pour construire des exécutables à partir des noeuds de couverture. Ce script utilise les outils de compilation et de génération fournis avec le langage LUSTRE.

2. *Un script d'exécution* pour lancer l'exécution des noeuds de couverture sur les séquences d'entrées qui lui sont fournis en entrée.
3. *Un script de récupération* pour restituer les résultats de la mesure sous forme de pourcentages de couverture.

LUSTRUCTU et mesure de la couverture

LUSTRUCTU est utilisé pour mesurer la couverture d'un programme écrit en LUSTRE. Le processus de mesure de la couverture est complètement automatisé. Comme illustré dans la figure 5.8, ce processus se décompose en quatre étapes récapitulées ci-dessous :

1. *Une phase de compilation* : A partir du programme sous test écrit en LUSTRE, LUSTRUCTU construit le modèle de couverture, à savoir le réseau d'opérateurs. A partir du réseau d'opérateurs et en fonction du critère de couverture choisi, le noyau de l'outil construit les noeuds de couverture associés à chaque critère. Un script permet, ensuite, de compiler et de produire les exécutables relatifs aux noeuds de couverture générés à l'étape précédente.
2. *Une phase de soumission* : Un deuxième script prépare et lance les séquences d'entrées. Les séquences d'entrées sont soit générées de manière aléatoire, soit préparées à partir de scénarios.
3. *Une phase d'exécution* : Une fois les noeuds de couverture exécutés avec comme entrées les séquences d'entrées, un script récupère le taux de couverture pour chaque critère. Ce script, permet également de récupérer l'ensemble des conditions d'activation qui ne sont pas satisfaites afin de les analyser.
4. *Une phase d'analyse et d'interprétation* des résultats récupérés après l'exécution. Cette phase est effectuée par l'intervention du testeur..

LUSTRUCTU et LUTESS

LUTESS est un outil de génération de données de test pour logiciels réactifs synchrones basé sur des techniques de test "boîte noire". Dans LUTESS, le logiciel

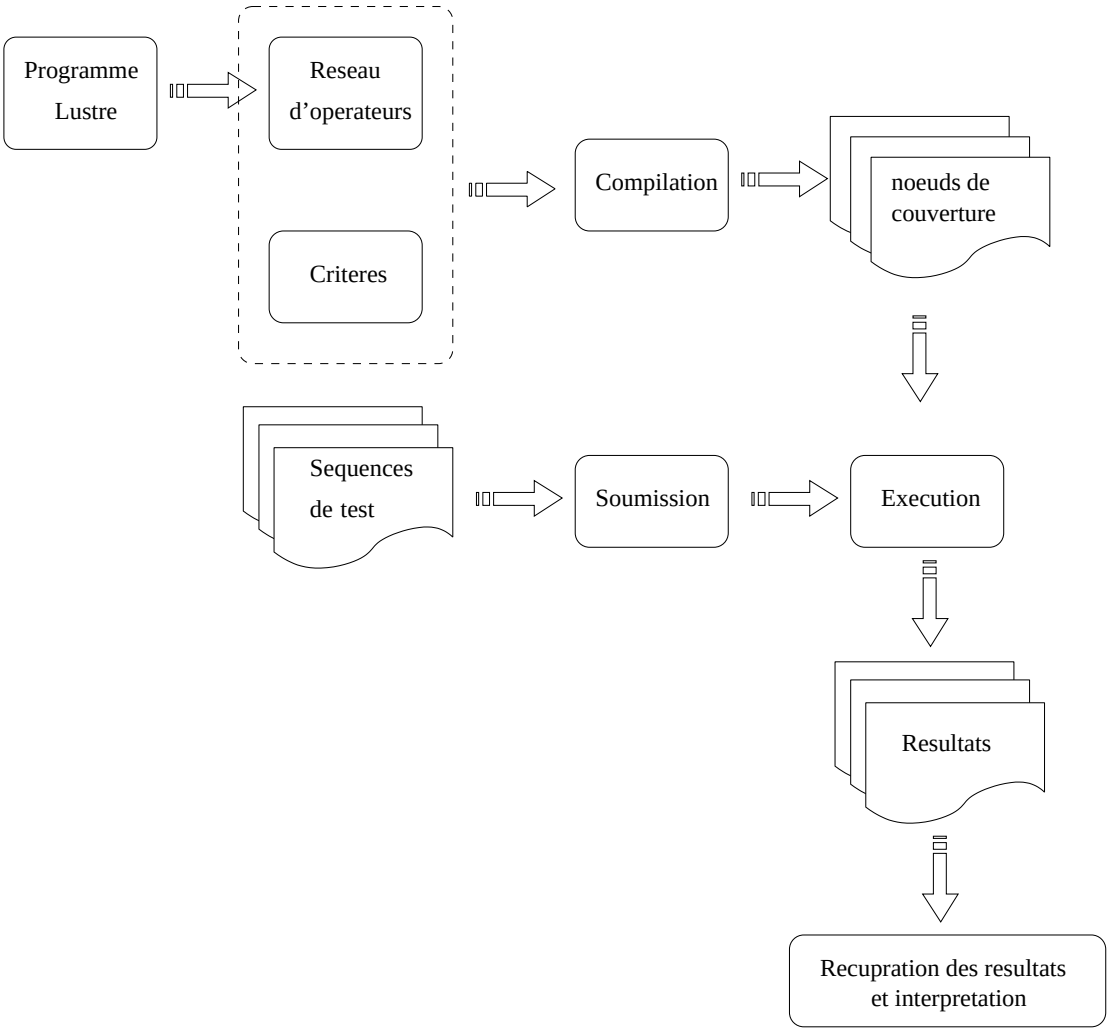

FIG. 5.8 – Mesure de la couverture par LUSTRUCTU

sous test est fourni sous forme binaire (exécutable). Dans le but de doter LUTESS de moyens pour mesurer la couverture structurelle du logiciel sous test par les séquences d'entrées générées à partir de spécifications fonctionnelles, une fonctionnalité "couverture" à base de LUSTRUCTU lui a été rajouté .

L'application des critères de couverture structurelle au logiciel sous test suppose que le programme sous test est un programme LUSTRE et que sa version source est disponible pour les testeurs. Par conséquent, l'interfaçage de LUTESS avec LUSTRUCTU nécessite que le code source LUSTRE du logiciel sous test soit connu. Dans sa version actuelle, LUTESS inclut une fonctionnalité de couverture qui a été utilisée pour mesurer la couverture de programmes dont les séquences d'entrées ont été générées en utilisant LUTESS.

L'une des utilisations possibles de LUSTRUCTU (envisagées en perspectives) est de couvrir les spécifications fonctionnelles d'un logiciel synchrone comme la spécification de l'environnement ou celle des propriétés de sûreté.

5.3 Étude de cas : contrôleur de l'alarme

Cette partie porte sur une évaluation des trois critères implémentés dans LUSTRUCTU (couverture de base, couverture des conditions élémentaires et couverture des conditions multiples). Plusieurs expérimentations ont été menées sur des exemples académiques et d'autres extraits d'études de cas industrielles. Nous présentons les résultats de l'une d'elles dans la suite de ce chapitre.

5.3.1 Objectifs de l'expérimentation

Les expérimentations que nous avons menées et dont nous présentons dans ce qui suit ont un triple objectif

1. Le premier objectif de cette étude est de montrer l'applicabilité des critères de couverture sur des programmes de taille et de complexité variées. A cet effet, LUSTRUCTU est utilisé pour automatiser le calcul des chemins et des

conditions d'activation associées, et observer ensuite le nombre de chemins calculés et les ressources système nécessaires. Une caractéristique principale des critères proposés est leur aptitude à évaluer de manière précise la progression de la couverture obtenue. En règle générale, ceci devrait signifier qu'une légère augmentation dans la satisfaction des critères devrait correspondre à peu d'effort additionnel du processus de test.

2. Par conséquent, le second objectif a été d'étudier dans quelle mesure l'effort requis pour satisfaire les critères dépend de la force du critère (le critère choisi et la longueur des chemins). L'effort est mesuré en termes de longueur de séquences d'entrées nécessaires à la satisfaction de chaque critère. Pour évaluer cet effort de manière non biaisée et neutre, les données de test (séquences d'entrées) utilisées ont été générées de façon aléatoire.

3. Le troisième et dernier objectif a été d'évaluer la pertinence des critères en termes de leur aptitude à détecter les fautes dans le programme sous test. En l'absence de "modèles de fautes", le test "mutationnel" a été utilisé pour simuler les fautes dans le programme. Un ensemble d'opérateurs de mutation a été défini et plusieurs mutants ont été construits de manière automatique. Les mêmes données de test (séquences d'entrées) ont été exécutées sur ces mutants pour calculer le score mutationnel (le taux des mutants tués) est ensuite calculé et comparé à la couverture obtenue par les critères.

5.3.2 Description de l'étude

L'étude de cas concerne un composant logiciel écrit en Lustre/Scade modélisant le fonctionnement d'une alarme dans un système de contrôle de vol. La fonction du composant est d'envoyer un signal d'alerte en fonction des paramètres de vol calculés par le système, de paramètres lus à partir de capteurs qui donnent l'état de l'appareil et de paramètres introduits par les pilotes. De manière sommaire et informelle, l'alerte est déclenchée lorsque la différence entre une valeur entrée par les

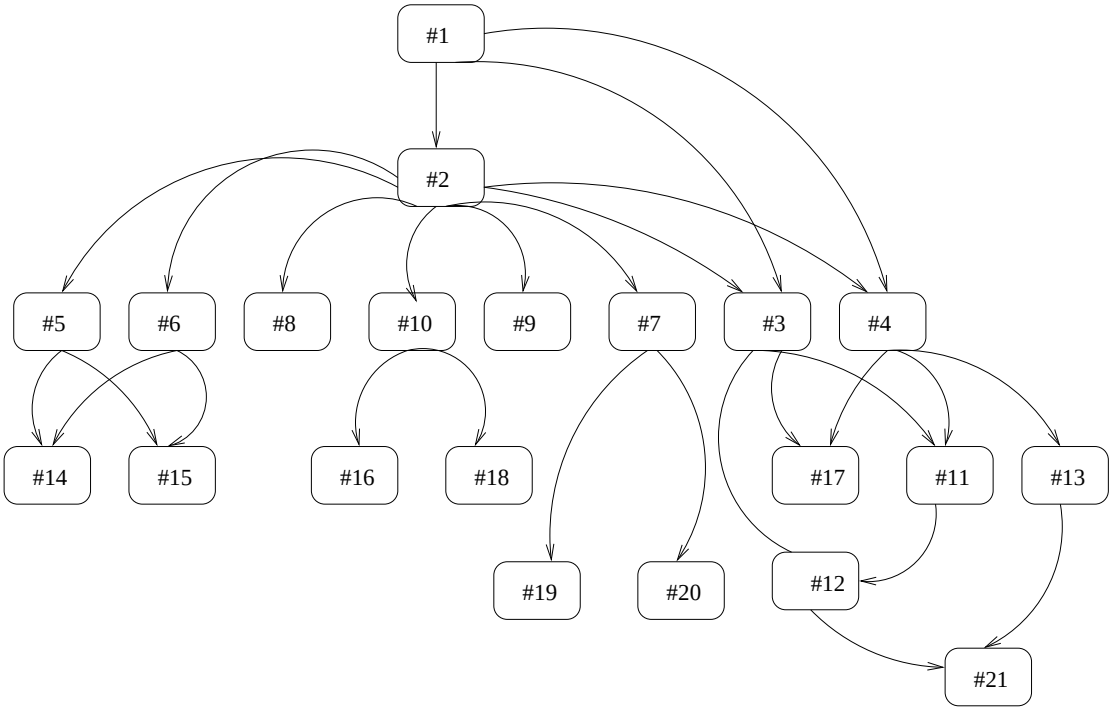

FIG. 5.9 – Graphe d'appel des modules de l'application

pilotes, la valeur théorique calculée à partir du centre de gravité et la valeur de position réelle dépasse 1.5°.

L'application est implémentée de façon modulaire. En effet, le composant est fourni sous forme d'une vingtaine de modules (noeuds) LUSTRE, générés à partir de l'environnement de développement LUSTRE/SCADE. L'application est constituée d'un module principal qui fait appel à des modules utilitaires selon le graphe d'appel de la figure 5.9.

Les données relatives à la taille et la complexité de l'application sont fournies dans la table 5.2. La table est divisée en deux parties :

1. Les trois premières colonnes concernent le code LUSTRE
   - *LOC* fournit le nombre de ligne de code LUSTRE dans chaque module.
   - *entrées* donne le nombre de variables d'entrées de chaque module
   - *sorties* donne le nombre de variables de sorties de chaque module
2. Les deux dernières colonnes sont associées aux réseaux d'opérateurs associés
   - *opérateurs* donne le nombre d'opérateurs

– *arcs* donne le nombre d'arcs

Les données de cette table correspondent à une version de l'application où tous les noeuds sont dépliés (avec uniquement des opérateurs de base). Le programme principal (module 1), par exemple, comprend 830 lignes de code Lustre. En entrées, ce noeud prend toutes les variables d'entrée du système (29 variables dont 23 booléennes et 6 entières). Fonctionnellement, ce module correspond à une propriété définie sur les deux alertes modélisées par les modules 2 et 3. Les autres noeuds de l'application (numérotés de 2 à 20 dans la table 5.2) sont appelés à partir du noeud principal et sont de complexité inférieure.

Les données de la deuxième partie de la table 5.2 récapitulent les caractéristiques des réseaux d'opérateurs associées à chacun des noeuds (modules) en termes de nombre d'arcs et d'opérateurs. Par exemple, le réseau d'opérateurs associé au noeud principal comprend 190 opérateurs reliés par 276 arcs. Une vue d'ensemble de ce noeud est illustrée par le réseau d'opérateurs de la figure 5.10. Pour la clarté du réseau, les noeuds composants ont été volontairement modélisés comme des boîtes noires avec leurs interfaces.

5.3.3 Exemple de programme

L'expérimentation que nous avons menée sur cette étude de cas a été effectuée sur tous les noeuds de l'application séparément. Sans perdre de généralité, les résultats que nous présentons, dans la suite de ce chapitre, concernent le noeud #2. Il s'agit d'un programme Lustre d'une taille moyenne (148 lignes de code Lustre).

Les calculs des expérimentations ci-dessous ont été effectués sur un système Linux Fedora (processeur Intel Pentium 2Ghz, 1024 Mo de RAM),

5.4 Nombre de chemins et de conditions d'activation

Le nombre de chemins dépend de l'ordre du critère. En revanche, le nombre de conditions d'activation qui doivent être activées pendant le test dépend de la force

Table 5.2: Tailles des noeuds de l'étude de cas

	Code			**Réseau**	
noeud	*LOC*	*entrées*	*sorties*	*arcs*	*opérateurs*
1	830	29	13	275	190
2	148	10	3	52	32
3	157	10	1	59	37
4	98	7	2	33	22
5	98	9	2	33	22
6	67	8	6	60	32
7	40	6	2	12	6
8	40	6	2	12	6
9	132	6	5	36	24
10	50	3	1	16	9
11	31	2	1	11	5
12	30	2	1	11	5
13	16	2	1	3	1
14	16	2	1	3	1
15	16	2	1	3	1
16	21	1	1	8	5
17	19	1	1	5	3
18	8	1	1	23	12
29	8	1	1	17	6
20	4	3	1	5	2

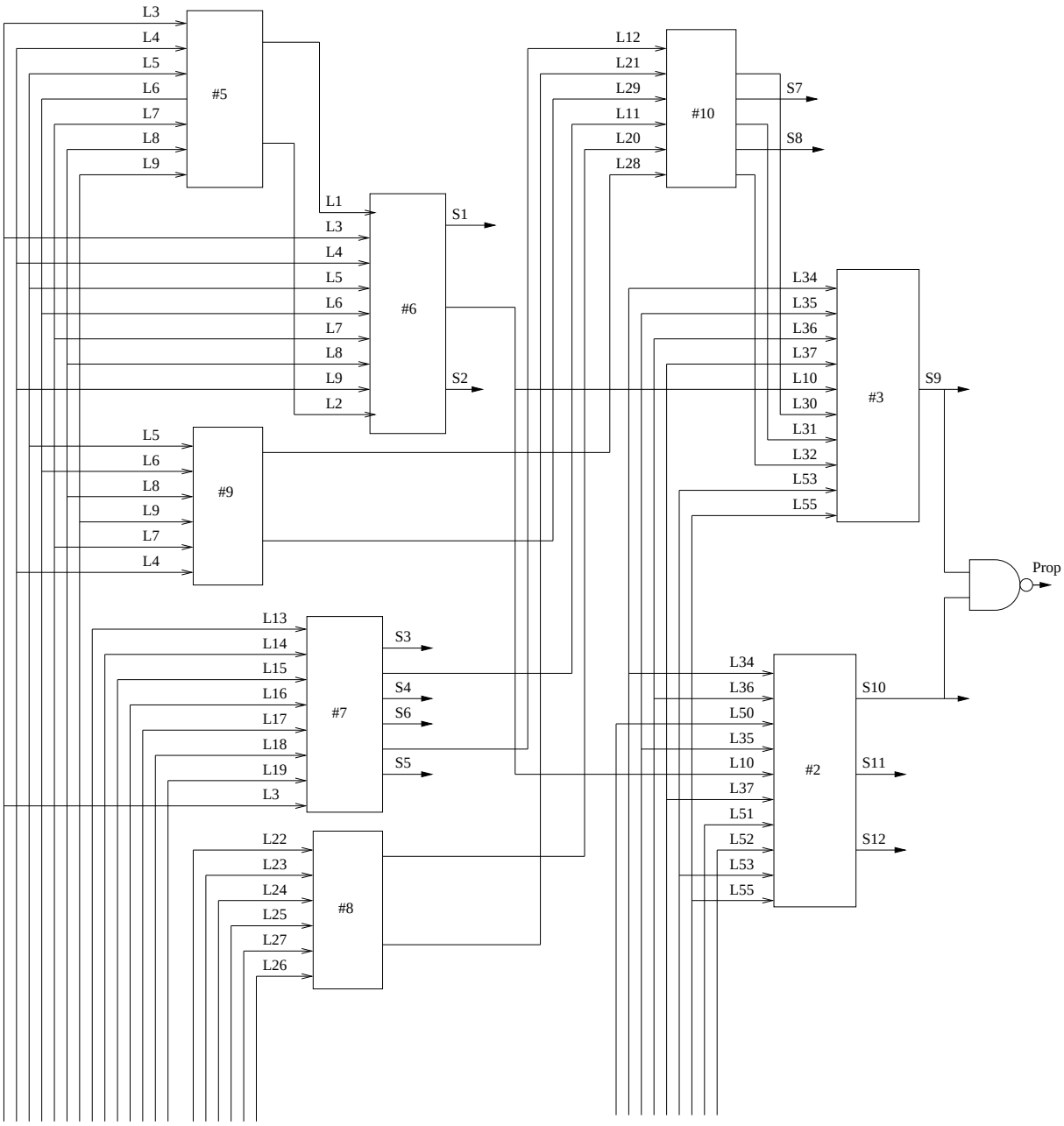

FIG. 5.10 – Réseau d'opérateurs (vue d'ensemble)

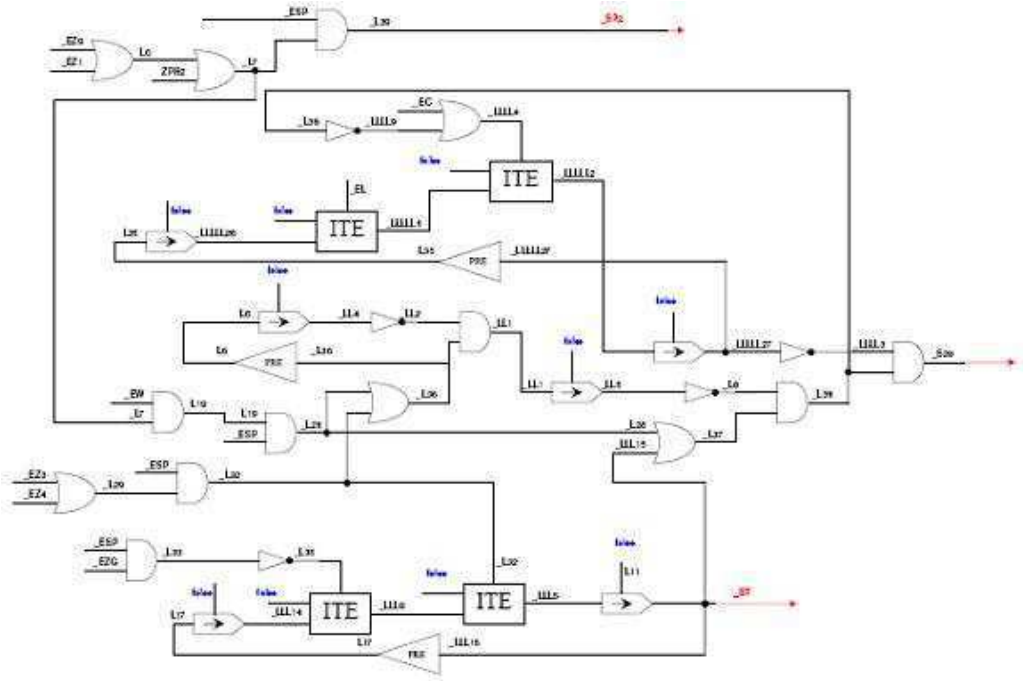

Figure 5.11: Réseau d'opérateurs du noeud (#2)

du critère (BC, ECC, MCC). Dans la pratique, nous ne considérons que des chemins complets (reliant une sortie à une entrée).

Nombre de chemins

Pour un ordre quelconque n, le nombre de chemins est le même pour les trois critères (BC, ECC, MCC). En effet, si n est un entier tel que $n \geq 2$ et si m_k dénote le nombre de chemins de longueur $k \leq n$ (m_2 chemins de longueur 2, m_3 chemins de longueur 3, ..., m_n chemins de longueur n), alors le nombre N de chemins associés à l'ordre n est donné par l'équation 5.1.

$$N = \sum_{k=2}^{k=n} m_k \tag{5.1}$$

Le nombre de chemins N dépend de deux facteurs:

- La taille du modèle (nombre d'arcs et nombre d'opérateurs) qui est proportionnelle au nombre de chemins. En effet, lorsque la taille du réseau augmente, le nombre de chemins augmente également. Cependant, les quelques exemples

que nous avons expérimentés ne permettent pas d'extrapoler une mesure du rapport entre le nombre de chemins et la taille du réseau. D'autre part, la taille n'est pas le seul facteur qui permet de déterminer le nombre de chemins mais il y a la complexité du modèle.

- La complexité du modèle en termes de présence de chemins ayant des cycles, dans la mesure où plus de cycles signifie plus de chemins. En effet, si plusieurs cycles sont pris en compte, la longueur des chemins augmente, et leur nombre aussi. Comme pour le premier facteur, la complexité seule ne permet pas de donner une idée sur le nombre de chemins. Les données de la table 5.3 illustrent la progression du nombre de chemins en fonction du nombre de cycles pris en compte.

Le temps nécessaire au calcul des chemins est relativement négligeable pour des modèles de taille et de complexité raisonnables. Cependant, ce nombre peut être très lent si la taille ou la complexité augmentent. Ce problème se pose davantage si tous les modèles sont dépliés. Une solution serait d'envisager des approximations avec une approche de test par intégration (voir 6.2).

Nombre de conditions d'activation

Si le nombre de chemins pour un ordre quelconque est le même, le nombre de conditions d'activation varie selon la force du critère. Un critère plus fort impose davantage de contraintes sur les chemins. Si le nombre de chemins pour un ordre quelconque n est N, le nombre de conditions d'activation M pour chaque type de critère est calculé comme suit:

- Critère de base (BC): $M_{BC} = N$
- Critère de couverture des conditions élémentaires (ECC) : $M_{ECC} = 2 * N$
- Critère de couverture des conditions multiples (MCC) : alors le nombre de conditions d'activation est donnée par l'équation 5.2.

Table 5.3: Variation nombre de chemins w.r.t nombre de cycles

Cycles	**Longueur**
0	680
1	903
2	1722
3	2763
4	3947
5	5273
6	6741
7	8351
8	10103
9	11997
10	14033

$$M = 2 * \sum_{k=2}^{k=n}((k-1) * m_k) \tag{5.2}$$

La table 5.4 montre le nombre de conditions d'activation pour des chemins complets pour deux noeuds de l'application. Les données obtenues correspondent à des chemins complets avec au maximum deux cycles. Cette limitation est donnée en guise de simplicité.

Le temps nécessaire pour calculer les conditions d'activation est relativement négligeable pour les critères BC et ECC (au plus quelques secondes ont été nécessaires pour calculer les chemins complets ayant au plus deux cycles et leurs conditions d'activation). En revanche, ce calcul devient de plus en plus important pour le critère MCC à cause du nombre de conditions d'activation qui devient très grand. Ceci pourrait sembler être un inconvénient pour le critère de conditions multiples (MCC), si des séquences d'entrées devraient être générées pour chaque condition. En fait, plusieurs conditions d'activation sont satisfaites lorsqu'une séquence d'entrées est exécutée.

Table 5.4: Nombre de conditions d'activation

noeud	*BC*	*ECC*	*ECC*
1	1722	3444	62052
2	59	118	1538

Conclusion

En bref, ces observations suggèrent que les critères proposés sont utilisables pour des programmes réels dans la mesure où les chemins considérés sont de longueur limitée. Naturellement, sur des programmes de taille importante, le temps de calcul de ces chemins peut augmenter en fonction de la limite de la longueur et du nombre de cycles considérés. La section suivante donne plus de détails sur le relation entre l'effort de test et le taux de satisfaction des critères.

5.5 Relation entre l'effort de test et la satisfaction des critères

L'effort exigé pour produire des séquences d'entrée satisfaisant un critère dépend de la technique utilisée pour choisir ces séquences d'une part et de testeurs humains d'autre part. Il est habituel (par exemple dans DO-178B) que les séquences d'entrées soient dérivées des besoins fonctionnels, indépendamment de la structure du programme. Par conséquent, il est difficile d'évaluer l'effort qu'un testeur devrait assurer pour construire des séquences d'entrée pur satisfaire un critère de couverture, puisque le choix des séquences d'entrées dépend fortement des facteurs humains. Il y a de plus une difficulté spécifique à obtenir des tests construits par des testeurs humains.

Construction des données de test

Afin de fournir une mesure de l'effort nécessaire, qui soit le plus indépendant possible des stratégies de génération et des facteurs humains, nous avons considéré que les séquences d'entrées sont aléatoirement générées, indépendamment de toute exigence fonctionnelle ou structurelle. Bien que le test aléatoire reste une technique peu productive et souvent coûteuse, elle reste neutre vis-à-vis des critères. Ainsi, nous pouvons l'utiliser pour comparer les différents critères sur une moyenne de jeux de tests. La moyenne est calculée sur un ensemble de campagnes de test.

Afin de s'assurer que les données de test générées pour chaque campagne sont différentes des autres, des germes différents sont ultilisés (ex : 1000, 2000, 3000,.....). Pour des besoins d'automatisation du processus de génération, nous procédons par une variation régulière. Une campagne de test correspond, donc, à différents tests avec des longueurs de séquences croissantes :

1. Des séquences de longueurs 1, 2, ...9,10
2. Des séquences de longueurs 10, 20,.....90,100
3. Des séquences de longueurs 100, 200,....,1000
4. et ainsi de suite.....

La génération de séquences d'entrées sur plusieurs cycles (longueurs croissantes) repose sur l'hypothèse que plus il est difficile de satisfaire un critère, plus la séquence d'entrées pour satisfaire le critère est longue.

Mesure de la couverture

Pour mesurer la couverture suivant les trois critères, les étapes suivantes sont exécutées de façon itérative :

1. Génération aléatoire des données de test (séquences d'entrées).
2. Exécution des noeuds de couverture (pour chaque critère) sur chacune des séquences d'entrées de chaque compagne.
3. Calcul de la moyenne des résultats obtenus.

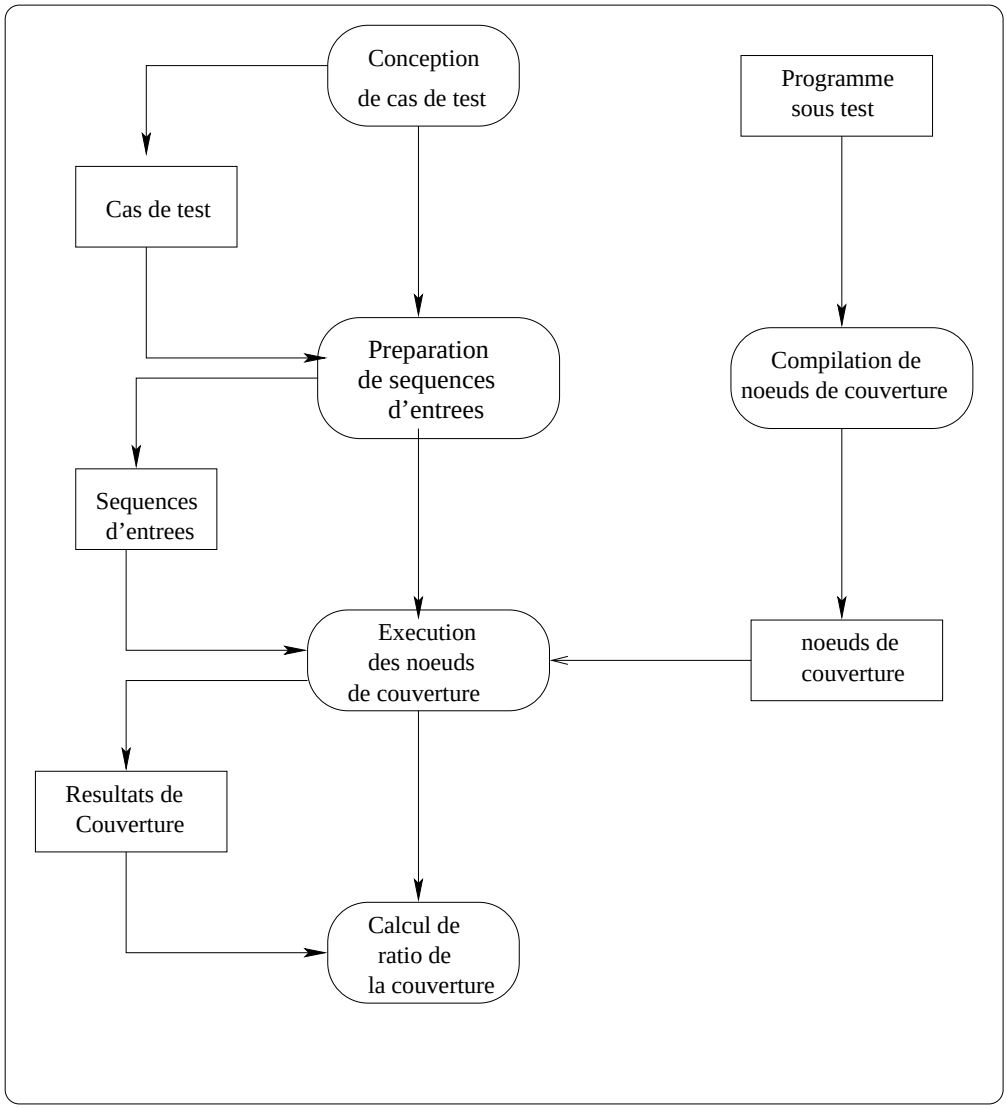

FIG. 5.12 – Processus de mesure de la couverture

Taux de satisfaction des critères

Les données des tables 5.5, 5.6 et 5.7 et les graphiques associés (voir Fig. 5.13, Fig. 5.14 et Fig.5.15 respectivement) illustrent la relation observée entre le taux de satisfaction des critères et la longueur des séquences d'entrées qui ont été nécessaires pour atteindre ces taux pour le noeud #2 (voir 5.2). Des résultats similaires ont été obtenus pour les autres noeuds.

Les barres dans ces graphiques représentent la variation du taux de couverture pour des séquences d'entrées dont la longueur varie respectivement de 1 à 10 (Fig. 5.13), de 10 à 100 (Fig. 5.14) et de 100 à 1000 (Fig. 5.15). Il est évident que le taux de

Table 5.5: Taux de couverture (séquences de longueur 1-10)

Longueur	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
BC	3,90	29,15	45,08	45,08	53,73	61,69	69,32	69,32	69,32	69,49
ECC	1,95	14,83	27,20	29,15	35,68	41,78	48,31	52,03	52,20	52,63
MCC	0,44	10,00	20,38	21,43	26,94	33,99	41,19	44,13	44,17	44,51

Table 5.6: Taux de couverture (séquences de longueur 10-100)

Longueur	*10*	*20*	*30*	*40*	*50*	*60*	*70*	*80*	*90*	*100*
BC	69,49	84,75	**100,00**							
ECC	52,63	69,24	82,37	86,02	89,24	89,49	89,83	89,83	89,83	89,83
MCC	44,51	57,24	74,36	81,10	82,72	83,47	84,19	84,54	84,78	84,78

la couverture, tous critères confondus, est proportionnel à la longueur des séquences d'entrées (hypothèse ci-dessus). En d'autres termes, plus les séquences sont longues plus le taux de couverture obtenu est élevé et vise et versa.

Le critère de couverture de base (BC) est satisfait relativement vite (après une séquence de longueur 30). En revanche, la satisfaction des critères de couverture de conditions élémentaires (ECC) et celui de couverture de conditions multiples (MCC) n'atteignent jamais 100%. La satisfaction de chacun de ces deux critères converge vers un seuil et la progression devient très minime. Par ailleurs, la satisfaction des trois critères est importante pendant les premières séquences et progresse relativement vite. La progression de la satisfaction décroît ensuite jusqu'à ce qu'elle atteint un niveau de stabilité qui diffère selon la force du critère.

Dans les trois graphes, les courbes illustrent le fait que, pour la génération aléatoire, le taux de couverture des trois critères croit de manière relativement rapide jusqu'à un certain niveau où il se stabilise tout en respectant une différence constante entre l'effort nécessaire pour satisfaire les trois critères.

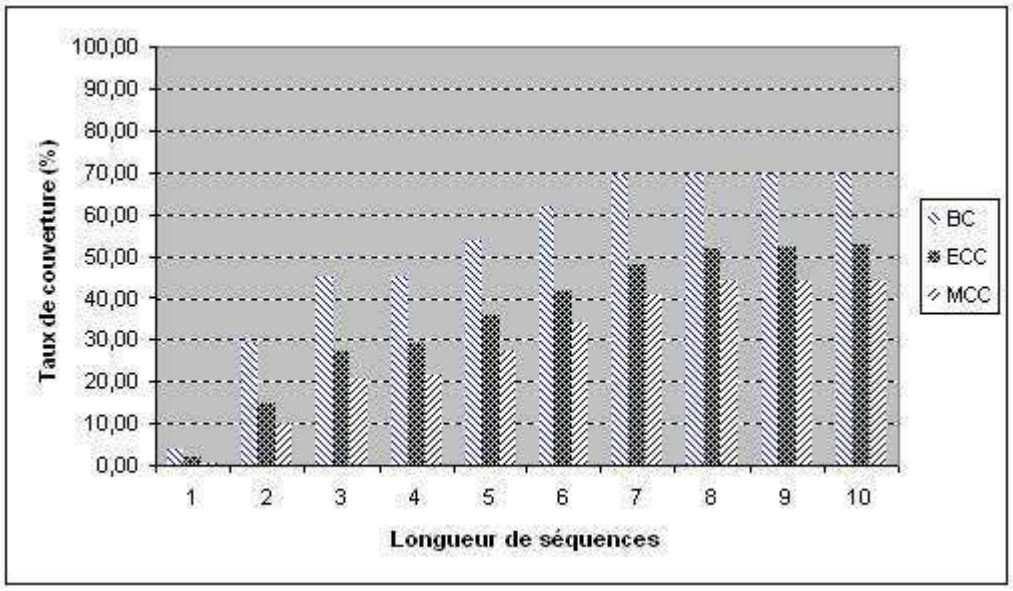

FIG. 5.13 – Séquences de 1 à 10

Il est à noter également le fait que le critère le plus faible est satisfait au bout de la 30ème séquence et que le taux de couverture des critères plus forts se stabilise autour de 93% (pour le critère de couverture des conditions élémentaires) et autour de 85% (pour le critère de couverture des conditions multiples). Ce qui s'explique, entre autres, par les limites de la génération aléatoire. Le reste de la couverture peut être complété par des jeux de test conçus par des spécialistes.

Par ailleurs, il y a une différence claire entre l'effort exigé pour satisfaire BC, ECC et MCC. En effet, atteindre le même niveau de satisfaction (même pourcentage) n'exige pas le même effort qu'il s'agisse de BC, d'ECC ou de MCC. Par exemple, pour atteindre 85% de satisfaction des trois critères, il a fallu générer des séquences de longueur 20 pour le critère de base, mais il a fallu augmenter cette longueur jusqu'à 40 pour le critère ECC et jusqu'à 200 pour le critère MCC.

Chemins infaisables

La non satisfaction à 100% d'un critère de couverture a deux raisons : la première est liée au jeu de tests qui n'est pas complet ; la seconde est indépendante du jeu de tests mais plutôt liée à la structure du programme et au critère (i.e., chemins

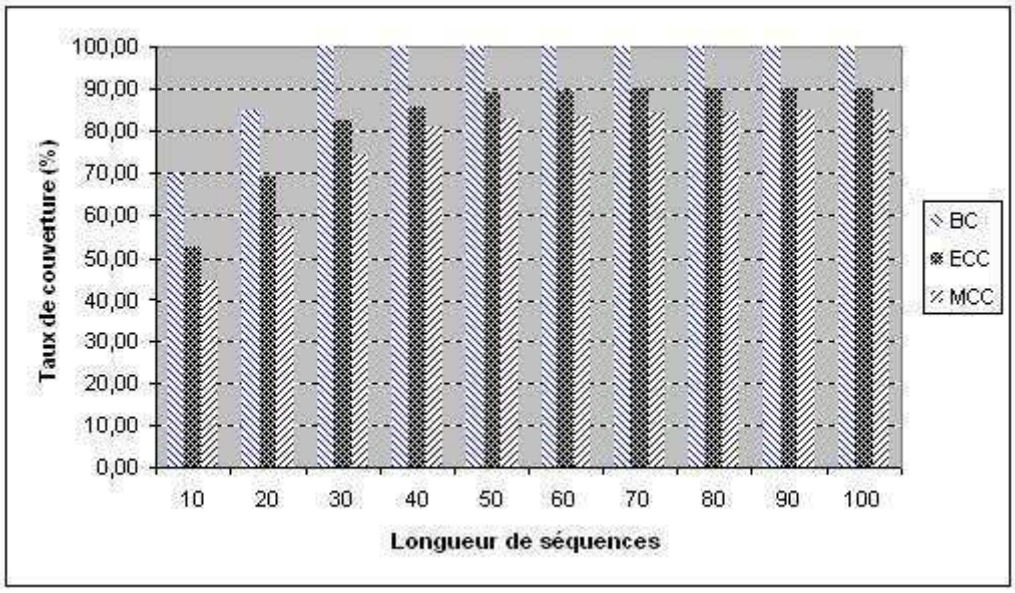

FIG. 5.14 – Séquences de 10 à 100

Table 5.7: Taux de couverture (séquences de longueur 100-1000)

Longueur	*100*	*200*	*300*	*400*	*500*	*600*	*700*	*800*	*900*	*1000*
ECC	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83
MCC	84,78	87,10	87,79	87,79	88,05	88,05	88,05	88,05	88,05	88,05

infaisables). La couverture complète du critère de couverture de base signifie qu'il ne peut y avoir de chemins infaisables au sens de la couverture de base. Ceci signifie que toutes les conditions d'activation ont été satisfaites au moins une fois.

En revanche, le critère de couverture des conditions élémentaires (ECC) et le critère de couverture des conditions multiples (MCC) ne sont pas satisfaits à 100% même pour des séquences très longues (quelques milliers). Ceci peut être dû à deux raisons : conditions infaisables ou à des configurations singulières nécessitant des combinaisons d'entrées rares.

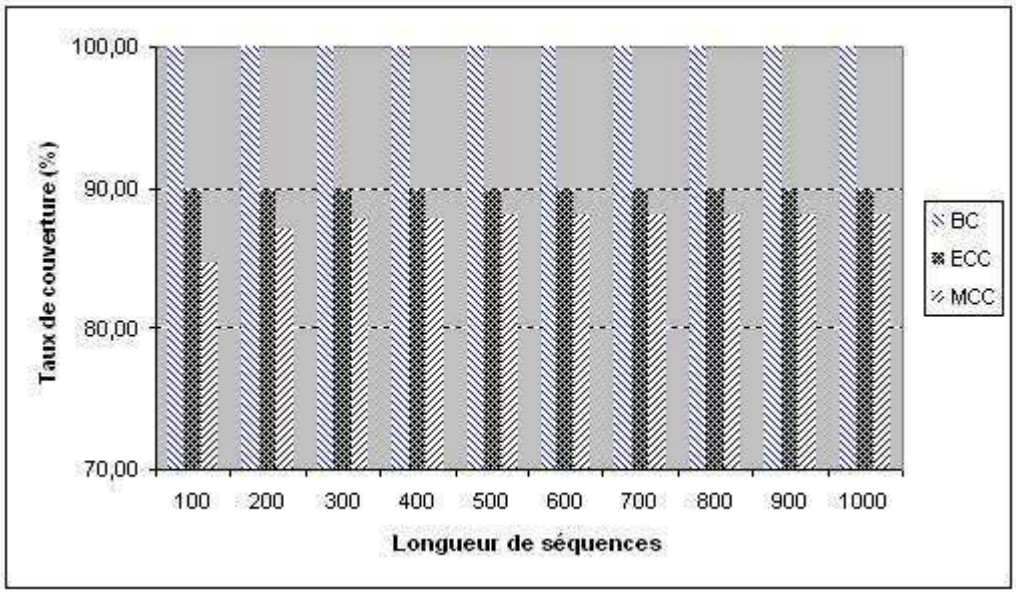

FIG. 5.15 – Séquences de 100 à 1000

Conclusion

Nous pouvons alors conclure qu'en règle générale, l'effort requis pour atteindre un rapport élevé de satisfaction ne serait pas très haut. Comme résultat de cette observation, un testeur humain devrait essayer d'abord de réaliser 100% BC. Ceci devrait également assurer un rapport élevé (70-80%) de satisfaction d'ECC et de MCC. Les conditions d'activation restantes (à 100% ECC et MCC, si possible) devrait exiger, probablement, une analyse du programme et auraient besoin probablement d'un effort de test plus important.

5.6 Aptitude à détecter des fautes

Modèle de fautes

Le test par mutation est un moyen habituel d'évaluer l'aptitude d'un jeu de tests à détecter des fautes. Contrairement au test des circuits électroniques où les testeurs disposent, souvent, de modèles de fautes "*métier*", nous nous disposons pas de modèles de fautes "*métier*" spécifiques aux programmes LUSTRE/SCADE. Nous avons considéré que des fautes peuvent être assimilées aux mutations sur des

Table 5.8: Opéateurs de mutation

Opérateur	*Remplacement*
NOT	PRE,empty
AND	OR, FBY
OR	AND, FBY
PRE	NOT, empty
Relationnel	Relationnel
Arithmétique	Arithmétique

Table 5.9: Nombre de mutants valides et non-équivalents

noeud	*mutants*
1	398
2	44
3	50
4	52

opérateurs.

Nous proposons une technique de mutation sélective. Dans cette technique, les opérateurs de mutation consistent à remplacer tout opérateur du langage par un autre de telle sorte que le programme obtenu (mutant) soit syntaxiquement correct.

Dans cette expérimentation, nous nous sommes limités aux mutations (opérateurs de mutation) présentées dans la table 5.8. Nous avons construit un générateur simple de mutants qui produit automatiquement des programmes mutants. A titre indicatif, la table 5.9 donne le nombre de mutants pour les quatre premiers noeuds. Ensuite, nous avons calculé le score mutationnel et l'avons comparé avec le taux de satisfaction des critères.

Table 5.10: Satisfaction des critères et score mutationnel (1-10)

Longueur	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
BC	3,90	29,15	45,08	45,08	53,73	61,69	69,32	69,32	69,32	69,49
ECC	1,95	14,83	27,20	29,15	35,68	41,78	48,31	52,03	52,20	52,63
MCC	0,44	10,00	20,38	21,43	26,94	33,99	41,19	44,13	44,17	44,51
Sc. mut.	11,36	18,18	18,18	18,18	29,55	59,09	63,64	63,64	68,18	68,18

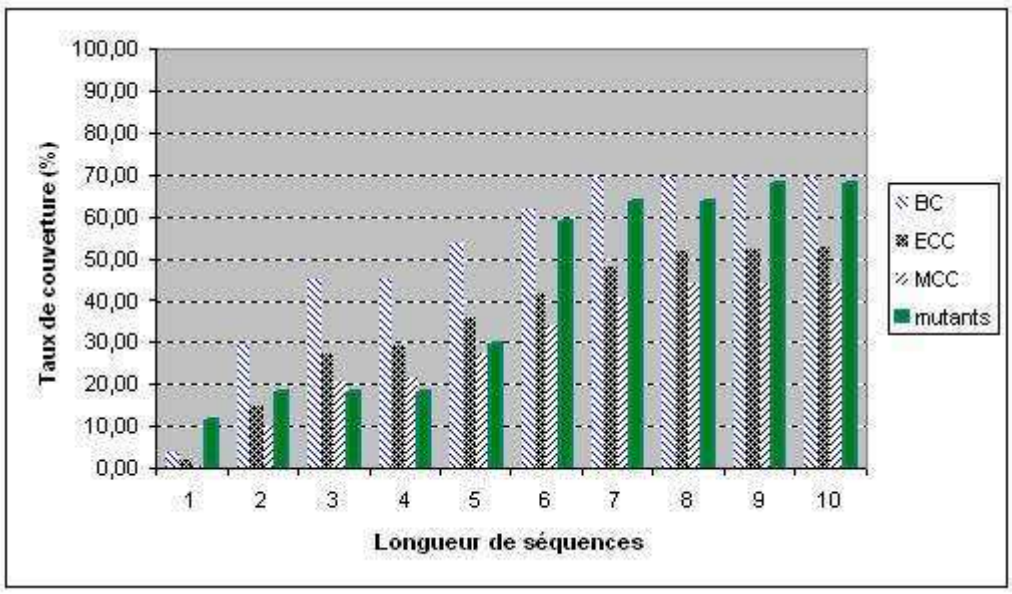

FIG. 5.16 – Score mutationnnel vs satisfaction des critères (1-10)

Observations et Conclusions

D'après les tables 5.10, 5.11 et 5.12 et les graphiques associés (voir Fig. 5.16, Fig. 5.17 et Fig. 5.18), il est facile d'observer qu'il y a une corrélation forte entre le score mutationnel et les taux de satisfaction des critères. Cette corrélation est proportionnelle à la force du critère. En effet, elle est beaucoup plus forte avec le taux de satisfaction du critère MCC qu'avec le critère ECC et davantage avec le critère BC. La satisfaction complète de ce dernier (100%) n'implique pas un score mutationnel élevé ce qui démontre que le critère est faible pour détecter les fautes usuelles dans un programme LUSTRE.

Table 5.11: Satisfaction des critères et score mutationnel (10-100)

Longueur	*10*	*20*	*30*	*40*	*50*	*60*	*70*	*80*	*90*	*100*
BC	69,49	84,75	**100,00**							
ECC	52,63	69,24	82,37	86,02	89,24	89,49	89,83	89,83	89,83	89,83
MCC	44,51	57,24	74,36	81,10	82,72	83,47	84,19	84,54	84,78	84,78
SC. mut.	68,18	77,27	79,55	86,36	86,36	86,36	86,36	90,91	90,91	95,45

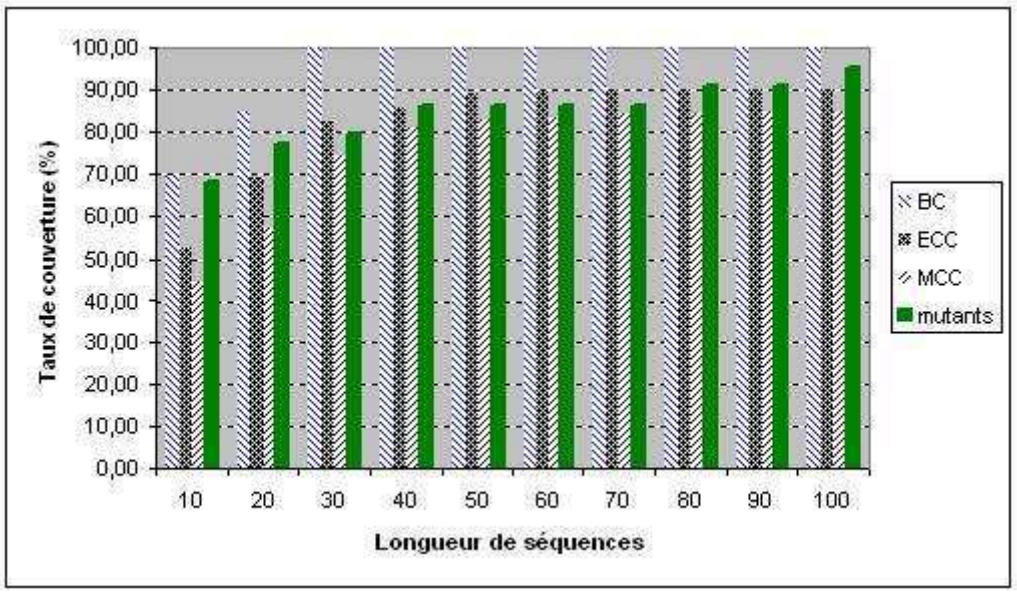

FIG. 5.17 – Score mutationnnel vs satisfaction des critères (10-100)

Table 5.12: Satisfaction des critères et score mutationnel (100-1000)

Longueur	*100*	*200*	*300*	*400*	*500*	*600*	*700*	*800*	*900*	*1000*
ECC	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83	89,83
MCC	84,78	87,10	87,79	87,79	88,05	88,05	88,05	88,05	88,05	88,05
Sc. mut.	97,73	**100,00**								

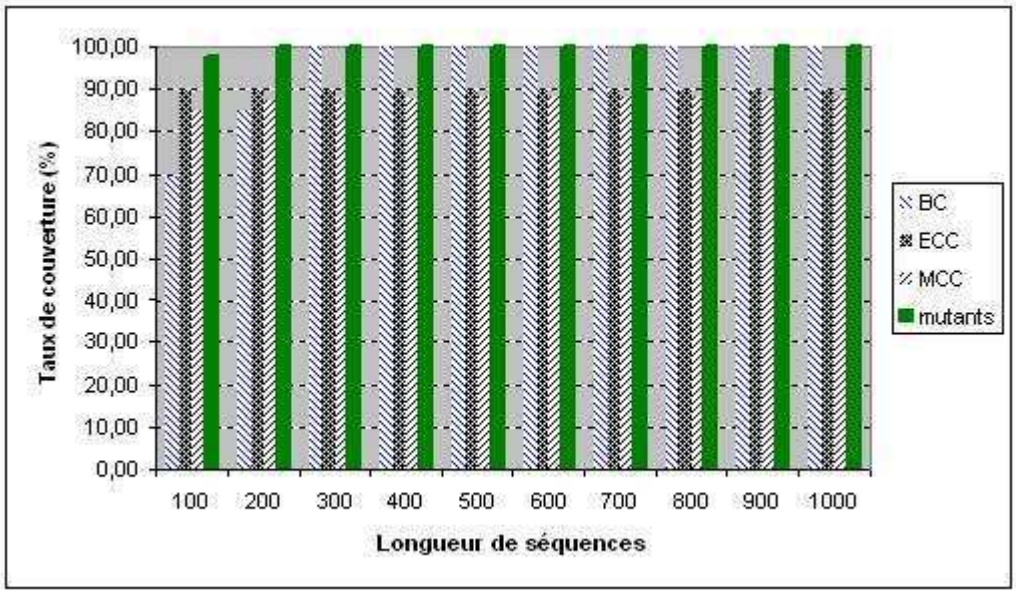

FIG. 5.18 – Score mutationnnel vs satisfaction des critères (100-1000)

5.7 Conclusions

L'objectif principal de la définition de ces critères de couverture était de fournir des métriques évaluant la qualité de jeux de test pour le code de LUSTRE/SCADE. Cette étude a montré que les critères BC, ECC et MCC (dont la force peut être augmentée en considérant des chemins de diverses longueurs) sont applicables en ce qui concerne le calcul des conditions d'activation. Cependant, le nombre de conditions d'activation MCC est plutôt élevé (plus de 60000) pour le noeud principal et il semble clair qu'un calcul manuel des tests est impraticable. Par conséquent, ces critères devraient être employés pour évaluer la qualité de code LUSTRE réalisée avec les tests qui sont calculés indépendamment des considérations de qualité.

Par ailleurs, les résultats de cette étude ont montré qu'il y a une corrélation forte entre le taux de couverture et les aptitudes de détection de fautes, assimilés aux scores mutationnels. Une meilleure validation de l'utilité des critères exige leur utilisation par testeurs humains réels.

Chapitre 6

Conclusions et travaux futurs

6.1 Bilan de la thèse

Le travail que nous venons de présenter s'inscrit dans le cadre de développement de techniques et d'outils formels pour la validation des programmes synchrones écrits en LUSTRE. Il vient enrichir et compléter une famille de techniques de génération de données de test en adjoignant des critères qui permettent de mesurer la couverture de ces programmes jusqu'alors non abordée.

Ce travail répond à des besoins émanant à la fois du monde académique et du monde de l'industrie. En effet, dans les domaines où LUSTRE/SCADE est utilisé pour la spécification et la programmation des applications critiques, la construction de jeux de tests est, souvent, effectuée pour satisfaire des objectifs définis à partir des besoins fonctionnels. Le but des critères de couverture structurelle est d'évaluer la qualité des tests et permettre de décider de l'arrêt du test.

La principale contribution de ce travail est la définition formelle de critères de couverture adaptés au langage LUSTRE [28, 29]. Cette définition tient compte de la nature "flot de données" du langage ainsi que des aspects temporels. En particulier, nous avons choisi un modèle de couverture adéquat, les réseaux d'opérateurs. La notion de couverture que nous avons introduite est définie sur des chemins du réseau de longueur finie, et s'appuie sur une forme de dépendance entre les données via ces chemins. Cette dépendance a été formalisée par la notion de condition d'activation

qui définit des contraintes minimales sur les chemins du réseau.

Une hiérarchie de trois critères a été proposée :

- Une couverture de base satisfaisant ces contraintes minimales
- Une couverture des conditions élémentaires satisfaisant les conditions d'activation avec variation des entrées des chemins.
- Une couverture des conditions multiples satisfaisant les conditions d'activation avec variation des arcs internes des chemins.

Ces critères ont été implémentés dans un outil appelé LUSTRUCTU, qui permet d'automatiser la mesure de la couverture en utilisant ces critères. L'outil a été utilisé pour mener des expériences sur une étude de cas du domaine de l'avionique. Les objectifs de cette étude ont consisté, entre autres, à montrer l'applicabilité des critères et à mesurer l'effort de test nécessaire à satisfaire chacun des critères et enfin à évaluer leur aptitude à révéler les fautes dans un programme LUSTRE. En l'absence de modèles de fautes "métier" spécifiques aux programmes LUSTRE, le test par mutation a été utilisé pour simuler des modèles de fautes.

Par ailleurs, l'utilisation de nos critères par des outils de génération comme GATEL pour renforcer les règles de dépliage des opérateurs a permis d'offrir des moyens de décomposition supplémentaires pour affiner et guider la génération. Les résultats de travaux menés conjointement [8] ont montré que la génération à base de ces critères pourrait être envisagée dans le futur.

Ces contributions devraient être utiles pour la validation des applications synchrones pour lesquelles des jeux de tests sont construits de manière manuelle ou à partir d'objectifs fonctionnels.

6.2 Perspectives et travaux futurs

Les critères de couverture définis dans le cadre de ce travail sont principalement destinés au test unitaire en boîte blanche, sur des noeuds de tailles raisonnables. A travers la formalisation et à la mise en oeuvre d'un processus d'automatisation de la mesure de couverture selon ces critères, le principal objectif était de mon-

trer l'applicabilité des critères et leur pertinence. De nombreuses questions restent ouvertes ; certaines constituent des perspectives immédiates et d'autres nécessitent une exploration plus approfondie. Parmi les questions ouvertes qui nécessitent plus d'exploration, nous citons les aspects liés :

- aux chemins infaisables et aux chemins difficiles à couvrir
- à la pertinence par rapport au test fonctionnel
- à l'optimisation.

Une étude approfondie des chemins infaisables et des chemins difficiles à couvrir permet d'identifier les parties d'un réseau d'opérateurs impossibles à couvrir et celles qui requièrent des combinaisons rares de cas de test et d'en donner une interprétation fonctionnelle. La pertinence des critères par rapport au test fonctionnel nécessite de nombreuses études de cas avec plusieurs types de programmes et plusieurs jeux de tests construits par des testeurs humains spécialistes du domaine.

Les questions d'optimisation concerneront par exemple la réduction de la taille des noeuds de couverture, en particulier pour le critère MCC. Des solutions basées sur le calcul symbolique sur les expressions booléennes peuvent amener à simplifier les conditions d'activation en identifiant d'éventuelles inclusions logiques (Une condition d'activation AC_1 est incluse dans une autre condition AC_2 si la satisfaction de la seconde implique la satisfaction de la première).

Les perspectives immédiates peuvent être envisagées dans deux directions : d'une part, la prise en compte des autres opérateurs de Lustre (opérateurs d'horloge) et d'autre part, le test d'intégration. Nous donnerons, dans la suite de cette section, quelques éléments qui permettent d'alimenter la réflexion autour de ces deux questions.

6.2.1 Prise en compte des autres opérateurs Lustre

Les opérateurs traités dans ce travail sont les opérateurs booléens, les opérateurs relationnels, les opérateurs arithmétiques et les deux opérateurs temporels les plus utilisés (PRE, FBY). Cependant, Lustre supporte d'autres opérateurs temporels, appelés opérateurs d'horloge, qui permettent de traiter des flots dans des horloges

distinctes.

Opérateurs d'horloge

Les opérateurs dits d'horloge sont l'opérateur *when* et l'opérateur *current*

- L'opérateur *when* permet d'échantillonner une expression selon une horloge plus lente : Si E est une expression et B est une expression booléenne avec une horloge quelconque, alors E *when* B est une expression dont l'horloge est définie par B et dont la séquence est extraite à partir de E en ne gardant que les valeurs des indices qui correspondent aux valeurs *true* dans la séquence B. En d'autres termes, c'est la séquence des valeurs de E lorsque B est vraie.
- L'opérateur *current* permet d'interpoler une expression sur l'horloge immédiatement la plus rapide. Soit E une expression dont l'horloge n'est pas l'horloge de base, et soit B l'expression booléenne qui définit cette horloge, l'expression *current* E a la même horloge C que B, et sa valeur à chaque instant de l'horloge C est la valeur de E au dernier instant où E était *true*.

B	$false$	$true$	$false$	$true$	$false$	$false$	$true$	$true$
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
$Y = X$ *when* B		x_2		x_4			x_7	x_8
$Z =$*current* Y	*nil*	x_2	x_2	x_4	x_4	x_4	x_7	x_8

Conditions d'activation

Afin d'étendre nos critères de couverture à des programmes contenant des opérateurs d'horloge *when* et *current*, le modèle de couverture (réseau d'opérateurs) est enrichi par deux autres noeuds WHEN et CURRENT respectivement. Nous donnons ci-dessous une première définition de la condition d'activation d'un chemin contenant l'un de ces deux opérateurs.

Soit $p = (e_1, ...e_{n-1}, e_n)$ un chemin de longueur n défini à partir de son préfixe $p' = (e_1, ...e_{n-1})$ et du chemin unitaire (e_{n-1}, e_n) telle que les deux arcs e_{n-1} et e_n sont inter-reliés par un opérateur *op* du type WHEN ou CURRENT.

De manière analogue aux définitions données au chapitre 4, la condition d'activation du chemin p est une expression booléenne temporelle :

$\mathcal{AC}(p) = \mathcal{AC}(p')\ and\ \mathcal{OC}(e_{n-1}, e_n)$

telle que $\mathcal{OC}(e_{n-1}, e_n)$ est définie comme suit :

1. Si op est un opérateur WHEN tel que $e_n = x\ when\ b$ alors :
   (a) $\mathcal{OC}(e_{n-1}, e_n) = b$ si $e_{n-1} = x$
   (b) $\mathcal{OC}(e_{n-1}, e_n) = true$ si $e_{n-1} = b$
2. Si op est un opérateur CURRENT tel que $e_n = current\ e_{n-1}$
   (a) $\mathcal{OC}(e_n, e_{n-1}) = e_{n-1}$

Ces définitions s'appuient sur les mêmes intuitions de la notion de condition d'activation évoquées au chapitre 4. La prochaine étape serait d'intégrer cette définition dans l'outil LUSTRUCTU.

6.2.2 Test d'intégration

Les critères définis dans ce travail sont des critères de couverture qui s'appliquent dans un cadre du test unitaire. Le langage LUSTRE est un langage modulaire. Les programmes sont souvent construits en utilisant des opérateurs de base (booléens, relationnels, temporels,....) mais aussi des appels à de noeuds externes. Dans la suite de cette section, nous donnons quelques éléments de solution pour prendre en compte les appels de noeuds dans la définition des critères de couverture.

Solutions envisageables et leurs limites

A priori, il y a deux solutions facilement envisageables pour couvrir un programme LUSTRE contenant des appels à des noeuds externes.

1. Considérer le noeud composant comme un opérateur simple avec des conditions de chemins unitaires toujours activés.
2. Déplier de façon systématique et totale tous les noeuds composants afin d'obtenir un programme simple. Dans ce cas, tout programme LUSTRE contenant

des appels à d'autres noeuds est déplié en remplaçant tous ces appels par le code qui leur correspond. Le programme résultant de l'expansion ne comprend que des expressions de base et le réseau d'opérateurs associé ne contient que des opérateurs de base (booléens, arithmétiques, temporels,...).

La première option est moins réaliste car elle ne tient pas compte de la complexité des noeuds composants. La couverture obtenue en négligeant des noeuds internes ne constitue pas une mesure pertinente du réseau d'opérateurs.

La seconde option est coûteuse car l'expansion est faite de manière syntaxique par une simple substitution. La conséquence est que la taille du réseau obtenu a tendance à s'accroître très vite. Il en est de même pour le nombre de conditions à satisfaire pour les critères que nous avons définis. Cette explosion est plus significative pour le critère de conditions multiples (MCC) dont les combinaisons s'accroissent de façon exponentielle avec la longueur et le nombre de chemins. En effet, la couverture d'un chemin de longueur k nécessite la satisfaction de $(2*(k-1))$ conditions d'activation.. Par conséquent la satisfaction d'un critère d'ordre n avec m_2 chemins de longueur 2, m_3 chemins de longueur 3,.....,m_n chemins de longueur n nécessite la satisfaction de $2 * (m_2 + 2 * m_3 + ... + (n-1) * m_n)$ conditions d'activation.

A titre indicatif, dans l'étude de cas de l'alarme (voir chapitre 5), le programme LUSTRE obtenu par expansion de tous les noeuds internes comprend 1720 chemins dont la longueur est inférieure à 30. Cela signifie que le nombre de conditions d'activation est égal à 1720. Le nombre de conditions à satisfaire pour le critère de couverture des conditions élémentaires est le double, i.e., 3440. En revanche, le nombre de conditions à satisfaire pour le critère de couverture des conditions multiples dépasse 60000, ce qui donne des noeuds de couverture de taille colossale, dont la compilation et l'exécution nécessite un temps non négligeable.

Orientations pour le test d'intégration

Afin d'éviter ces deux solutions pour la prise en compte des noeuds dans la couverture des réseaux d'opérateurs, l'idée est de construire une approximation de la couverture des noeuds composés. Cette approximation serait un compromis entre

la couverture issue de l'expansion totale des noeuds et la couverture minimale qui consiste à considérer qu'un noeud est toujours couvert. De cette manière, on obtiendrait une couverture réaliste tout en évitant les problèmes d'explosion combinatoire.

Le principe de l'approximation que nous proposons consiste à construire une abstraction du noeud appelé, basée sur un nouveau type d'opérateurs multi-entrées multi-sorties : NODE.

Opérateur NODE

NODE est un opérateur dont le nombre d'entrées et de sorties est variable. Il est utilisé en lieu et place d'un appel à un noeud. Il dénote le réseau d'opérateurs associé à ce noeud où :

- Les entrées de l'opérateur sont les entrées du noeud remplacé.
- Les sorties de l'opérateur sont les sorties du noeud remplacé.
- L'abstraction est construite de telle sorte que l'ensemble des chemins reliant un arc d'entrée e_i à un arc de sortie s_j dans le réseau d'opérateurs du noeud remplacé soit assimilé à un seul chemin unitaire (e_i, s_j) dont la condition d'activation est une combinaison des conditions d'activation de l'ensemble de chemins (voir ci-dessous).

De manière analogue à tous les opérateurs de base (opérateurs booléens, opérateurs relationnels et opérateurs temporels), nous associons une condition à chacun des chemins unitaires de l'opérateur NODE. Ces conditions sont calculées comme suit :

Soit un opérateur NODE ayant un ensemble d'entrées E et un ensemble de sorties S et tel qu'il existe :

- une entrée $e_i \in E$
- une sortie $s_j \in S$
- k chemins $p_1, ..., p_k$ entre l'entrée e_i et la sortie s_j dont les conditions d'activation sont $AC_1, ..., AC_k$ respectivement.

La condition d'activation du chemin unitaire $p = (e_i, s_j)$ est la disjonction des conditions d'activation des k chemins qui relient l'entrée e_i à la sortie s_j (voir l'équation 6.1)

$$AC(p_{1n}) = AC_1 \vee AC_2 \vee ..... \vee AC_k \tag{6.1}$$

Il est évident que la condition d'activation traduit toujours une forme de dépendance entre l'entrée et la sortie. A minima, cette dépendance est définie comme l'une des dépendances possibles entre l'entrée et la sortie.

Bibliographie

[1] Scade language reference manual. technical report SC-LRM - SC/70257u3-5.0.1, Esterel Technologies SA, Parc Avenue, 9 rue Michel Labrousse, 31100 Toulouse, France, 2005.

[2] Vinod K. Agarwal and Andy S. F. Fung. Multiple fault testing of large circuits by single fault test sets. *IEEE Trans. Computers*, 30(11) :855–865, 1981.

[3] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, september 1991.

[5] A. Benveniste, G. Berry, P. Caspi, Ph. Couronné, F. Dupont, Th. Gautier, N. Halbwachs, P. Le Guernic, C. Le Maire, J.-P. Mignard, F. Paris, and Sorel Y. Synchronous technology for real-time systems. In *The 1994 Real-Time Conferences*, pages 104–122, Teknea, 1994.

[6] G. Berry. *The Foundations of Esterel.* MIT Press, 2000.

[7] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[8] Benjamin Blanc, Guy Durrieu, Abdesselam Lakehal, Odile Laurent, Bruno Marre, Ioannis Parissis, Christel Seguin, and Virginie Wiels. Automated functional test case generation from data flow specifications using structural coverage criteria. In *3rd European Congress on Embedded Real Time Software (ERTS2006)*, Toulouse, France, January 2006.

[9] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and emperical studies on using program mutation to test the functional correctness of programs. In *POPL*, pages 220–233, 1980.

[10] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[11] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11) :1318–1332, 1989.

[12] Richard A. DeMillo. Test adequacy and program mutation. In *International Conference on Software Engineering*, pages 355–356, 1989.

[13] Richard A. DeMillo. Progress toward automated software testing. In *International Conference on Software Engineering*, pages 180–183, 1991.

[14] Lydie Du Bousquet. *Test fonctionnel statistique de systémes spécifié en Lustre.* PhD thesis, Grenoble, France, Septembre 1999.

[15] Lydie du Bousquet and Nicolas Zuanon. An overview of lutess : A specification-based tool for testing synchronous software. In *Automated Software Engineering*, pages 208–215, 1999.

[16] Joe W. Duran and Simeon C. Ntafos. A report on random testing. In *International Conference on Software Engineering*, pages 179–183, 1981.

[17] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10) :1483–1498, 1988.

[18] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2) :156–173, 1975.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, september 1991.

[20] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9) :785–793, 1992.

[21] Richard G. Hamlet. Theoretical comparison of testing methods. In *Symposium on Testing, Analysis, and Verification*, pages 28–37, 1989.

[22] David Harel. Statecharts : A visual formulation for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, 1987.

[23] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[24] J. Ullman J. Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Compagny, 1979.

[25] S. Rao Kosaraju. Analysis of structured programs. *J. Comput. Syst. Sci.*, 9(3) :232–255, 1974.

[26] Abdesselam Lakehal, Farid Ouabdesselam, Ioannis Parissis, and Jérôme Vassy. Models for synchronous software testing. In *Proceedings of the Sivoes-MoDeVa Workshop : a satellite event of the 15th IEEE International Symposium on Software Reliability Engineering*, pages 41–50, Rennes, France, November 2004.

[27] Abdesselam Lakehal and Ioannis Parissis. Lustructu : A tool for the automatic coverage assessment of lustre programs. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 301–310, Chicago, Illinois, USA, November 2005.

[28] Abdesselam Lakehal and Ioannis Parissis. Structural test coverage criteria for lustre programs. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems : a satellite event of the ESEC/FSE'05*, pages 35–43, Lisbon, Portugal, September 2005.

[29] Abdesselam Lakehal, Ioannis Parissis, and Lydie Du-Bousquet. Critères de couverture structurelle des programmes lustre. In *Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 185–199, Besançon, France, Juin 2004.

[30] Jean-Claude Laprie. *Guide de la Sûreté de Fonctionnement.* Cépaduès, 1995.

[31] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, 9(3) :347–354, 1983.

[32] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9) :1321–1336, september 1991.

[33] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions : Gatel. In *Automated Software Engineering*, pages 229–237, 2000.

[34] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electr. Notes Theor. Comput. Sci.*, 111 :93–111, 2005.

[35] C. Mazuet. Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre. Thesis, Institut National Polytechnique de Toulouse, Toulouse, France, december 1994.

[36] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4) :308–320, 1976.

[37] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[38] Brian T. Murray and John P. Hayes. Testing ics : Getting to the core of the problem. *IEEE Computer*, 29(11) :32–38, 1996.

[39] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2) :14–32, 1993.

[40] G. J. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[41] Simeon C. Ntafos. An evaluation of required element testing strategies. In *International Conference on Software Engineering*, pages 250–256, 1984.

[42] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6) :868–874, june 1988.

[43] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *ICECCS*, pages 119–, 1999.

[44] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering*, Monterey, USA, november 1994.

[45] Amit M. Paradkar, Kuo-Chung Tai, and Mladen A. Vouk. Specification-based testing using cause-effect graphs. *Ann. Software Eng.*, 4 :133–157, 1997.

[46] Ioannis Parissis. *Test de logiciels synchrones spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, Septembre 1996.

[47] Ioannis Parissis. Vers une approche mixte de validtion des logiciels synchrones : preuve, test, animation. In *Approches Formelles dans l'Assistance au Développement de Logiciels*, Toulouse, France, may 1997.

[48] Ioannis Parissis and Farid Ouabdesselam. Specification-based testing of synchronous software. In *ACM-SIGSOFT Foundations of Software Engineering*, pages 127–134, 1996.

[49] David Parnas, John van Schouwen, and Shu Po Kwan. Evaluation of Safety-Critical Software. *Communications of the ACM*, 33(6) :636–648, june 1990.

[50] Daniel Pilaud and Nicolas Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *FTRTFT*, pages 99–110, 1988.

[51] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4) :367–375, 1985.

[52] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.

[53] Kuo-Chung Tai. Predicate-based test generation for computer programs. In *International Conference on Software Engineering '93 : Proceedings of the 15th international conference on Software Engineering*, pages 267–276, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[54] Pascale Thévenod-Fosse, Christine Mazuet, and Yves Crouzet. On statistical structural testing of synchronous data flow programs. In *EDCC*, pages 250–267, 1994.

[55] Jan Tretmans. A formal approach to conformance testing. In *Protocol Test Systems*, pages 257–276, 1993.

[56] Jan Tretmans, Pim Kars, and Ed Brinksma. Protocol conformance testing : A formal perspective on iso is-9646. In *Protocol Test Systems*, pages 131–142, 1991.

[57] Jérôme Vassy. *Génération automatique de cas de test guidée par les propriétés de sûreté*. PhD thesis, Université Joseph Fourier, Grenoble, France, october 2004.

[58] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the z notation. In *COMPSAC*, pages 351–356, 2001.

[59] Sergiy A. Vilkomir and Jonathan P. Bowen. Reinforced condition/decision coverage (rc/dc) : A new criterion for software testing. In *ZB*, pages 291–308, 2002.

[60] Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM J. Comput.*, 8(4) :587–598, 1979.

[61] L. White and E. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, pages 247–257, may 1980.

[62] Martin R. Woodward, David Hedley, and Michael A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Software Eng.*, 6(3) :278–286, 1980.

[63] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4) :366–427, 1997.

[64] Nicolas Zuanon. *Une méthode de test pour la validation de services de télécommunication*. PhD thesis, Grenoble, France, Juin 2000.

Annexe A

Compilation en boucle simple

Ci-dessous, le code C produit par compilation du noeud "never" en boucle simple (compilateur Lustre V4)

```
/********
* ec2c version 0.4-beta * c file generated for node : never
********/
#include <stdlib.h>
#define _never_EC2C_SRC_FILE
#include "never.h"
/*--------
  Internal structure for the call
--------*/
typedef struct {
    void* client_data;
   //INPUTS
    _boolean _A;
   //OUTPUTS
    _boolean _never_A;
   //REGISTERS
    _boolean M6;
    _boolean M6_nil;
    _boolean M2;
} never_ctx;


/*--------
Reset procedure
--------*/
void never_reset(never_ctx* ctx){
  ctx->M6_nil = _true;
  ctx->M2 = _true;
  never_reset_input(ctx);
}
/*--------
Dynamic allocation of an internal structure
--------*/
never_ctx* never_new_ctx(void* cdata){
  never_ctx* ctx = (never_ctx*)calloc(1, sizeof(never_ctx));
  ctx->client_data = cdata; never_reset(ctx);
  return ctx;
}
/*--------
Copy the value of an internal structure
--------*/
void never_copy_ctx(never_ctx* dest, never_ctx* src){
  memcpy((void*)dest, (void*)src, sizeof(never_ctx));
}
```

```
/*--------
Output procedures must be defined, Input procedures must be used :
--------*/
void never_I_A(never_ctx* ctx, _boolean V){
  ctx->_A = V;
}
extern void never_O_never_A(void*, _boolean);
#ifdef CKCHECK
extern void never_BOT_never_A(void*);
#endif
/*--------
Internal reset input procedure
--------*/
static void never_reset_input(never_ctx* ctx){
  //NOTHING FOR THIS VERSION...
}



/*--------
Step procedure
--------*/
void never_step(never_ctx* ctx){
  //LOCAL VARIABLES
  _boolean L4;
  _boolean L5;
  _boolean L1;
  _boolean T6;

  //CODE
  L4 = (! ctx->_A);
  L5 = (L4 && ctx->M6);
  if (ctx->M2) {
    L1 = L4;
  }
  else {
    L1 = L5;
  }
  never_O_never_A(ctx->client_data, L1);
  T6 = L1;
  ctx->M6 = T6;
  ctx->M6_nil = _false;
  ctx->M2 = ctx->M2 &&!(_true);
}
```