Utilisation des Structures Combinatoires pour le Test Statistique

Sandrine-Dominique Gouraud

N° d'ordre: 7550

THÈSE

Présentée devant

devant l'université de Paris-Sud-Orsay

pour obtenir

le grade de : Docteur de l'université de Paris-Sud-Orsay
Mention Informatique

par

Sandrine-Dominique GOURAUD

Équipe d'accueil : Programmation et Génie Logiciel
École doctorale d'Informatique
Composante universitaire : L.R.I.

Titre de la thèse :

Utilisation des structures combinatoires pour le test statistique

soutenue le 24 juin 2004 devant la commission d'examen

Présidente:	Pascale	Thévenod-Fosse
Rapporteurs:	Philippe	Flajolet
	Farid	Ouabdesselam
Examinateurs:	Alain	Denise
	Marie-Claude	Gaudel
	Bruno	Marre

Mon ordinateur, j'essaie de faire tout ce qu'il me dit
mais lui il fait rien de ce que je veux.

Anne Roumanoff, *Internet*

Remerciements

Je tiens à remercier chaleureusement Marie-Claude Gaudel, Alain Denise et Bruno Marre pour m'avoir encadrée, soutenue et conseillée durant ma thèse.

Je tiens aussi à remercier :

- Philippe Flajolet, directeur de recherche à l'INRIA, et Farid Ouabdesselam, professeur à l'Université Joseph Fourier, d'avoir accepté de relire et de juger mes travaux,
- Pascale Thévenod-Fosse, directrice de recherche au LAAS, pour avoir accepté de présider mon jury,
- Hélène Waeselynck et Pascale Thévenod-Fosse pour m'avoir fourni les programmes et mutants utilisés pour mes expériences,
- Sylvie Corteel et Nicolas Thiéry pour leur soutien technique sur leurs outils de génération aléatoire de structures combinatoires,
- l'équipe administrative et technique du LRI pour leur grande disponibilité et en particulier Dominique et Martine pour leur dévouement auprès des étudiants,
- l'équipe Programmation et Génie Logiciel pour leur soutien et leurs encouragements, et plus particulièrement, Fatiha et Eric pour leur bonne humeur, Gregory pour sa patience et surtout Frédéric pour sa patience mais aussi pour les différentes lectures, relectures et corrections de mes écrits,
- la bande des thésards, et en particulier Mathieu, syp et Thomas pour tous nos échanges très constructifs.

Plus personnellement, je tiens à dire un grand MERCI pour leur soutien sans faille et tous leurs encouragements à toute ma famille au sens large ainsi qu'à tous mes "potos" : Juju et Sandra, Draven et Marie-Anne, JP et Gaëlle, Guillaume, Alex, Marc et Cathy, Alex et Caro, Seb et Sandrine, le scout, Julien, Nathalie...

Enfin, je voudrais terminer cette page en remerciant ceux qui se sont le plus sacrifiés pour que j'en arrive là : mes parents et surtout mon Manu dont la patience, la tendresse et l'amour font de lui ma plus précieuse armure et muse.

Table des matières

Introduction

Cette thèse propose une nouvelle approche pour le test statistique de logiciel basée sur la génération aléatoire de chemins dans un graphe. Elle utilise des résultats théoriques sur la manipulation d'objets combinatoires issus du domaine algorithmique. Cette approche est générale c'est-à-dire applicable à n'importe quelle méthode de test statistique.

Une première application a été effectuée pour le test statistique structurel et un prototype a été développé. Ce prototype appelé AuGuSTe a permis l'exécution d'expériences permettant d'évaluer et de valider notre approche.

Ce premier chapitre rappelle le contexte actuel du test de logiciel, puis expose nos motivations ainsi que brièvement notre contribution. Il se termine sur un plan de ce mémoire.

Le test de logiciel

Contexte Aujourd'hui, les ordinateurs font de plus en plus partie de notre société : de la simple calculatrice aux systèmes électroniques des voitures (ABS, etc.) [11] en passant par la gestion des centrales nucléaires et le pilotage des avions, etc. Chacun de ces ordinateurs se différencie par le ou les logiciels qui le "commande(nt)". Du cahier des charges à l'intégration du logiciel, le processus de création de ces logiciels est long et complexe. Pour assurer que le produit est conforme aux objectifs attendus, il faut valider chaque étape du développement. Si un bogue dans la version finale du logiciel est, au niveau de l'utilisateur moyen, relativement négligeable (pertes de données personnelles) dans le sens où ce ne sont seulement finalement que des heures de travail perdues, il n'en est pas de même dans d'autres domaines.

En effet, dans le cas des logiciels critiques, comme ceux qui dirigent les centrales nucléaires ou encore les systèmes embarqués des avions, trains et autres transports, les conséquences d'une erreur peuvent être beaucoup plus graves car ce sont des catastrophes humaines ou naturelles qui sont en jeu. La fiabilité de ces logiciels est très importante, c'est pourquoi des chercheurs, comme des industriels, travaillent au développement de méthodes efficaces et surtout d'outils pour

automatiser la partie tests du développement des logiciels [52].

Le test de logiciels n'est pas un procédé facile à mettre en place car en plus du coût en temps et en argent, la complexité de ces logiciels devient de plus en plus importante. En mai 2002, une étude[1] menée pour le NIST [94] sur l'impact de l'insuffisance d'infrastructure de test dans les développements de logiciels a montré que le coût de cette insuffisance est d'environ 60 milliards de dollars pour l'économie américaine [2, 1]. Il faut donc développer de nouvelles méthodes de tests automatisables.

L'objectif de cette thèse est la définition d'une méthode de test de logiciel complètement automatisée.

Les différents types de test Dans le processus de développement du logiciel, il existe différents types de test dont :

- les *tests unitaires* qui permettent de tester les composants un par un.
- les *tests d'intégration* qui permettent de tester l'ajout d'un nouveau composant (testé) dans un ensemble de composants.
- les *tests "système"* qui permettent de tester le système sur son futur site d'exploitation, dans des conditions opérationnelles et au-delà[2] (surcharge, défaillance matérielle, etc.) .
- les *tests de régression* qui permettent de vérifier que la modification d'un ou de plusieurs composants n'entraîne pas d'incohérence entre la nouvelle version et la précédente.

Dans ce mémoire, on s'intéresse aux tests unitaires.

Généralement tester un logiciel consiste à sélectionner des entrées pour ce logiciel et à vérifier que le résultat et/ou le comportement obtenu est conforme à ce qui était attendu. Il existe de nombreuses méthodes pour sélectionner des ensembles de test. Classiquement, on répartit ces méthodes en trois familles : les méthodes fonctionnelles qui sélectionnent l'ensemble des entrées à partir de la spécification du programme à tester, les méthodes structurelles qui sélectionnent cet ensemble à partir de la structure (instructions, enchaînements, chemin d'exécution, etc.) du programme et les méthodes statistiques qui sélectionnent cet ensemble à partir d'une distribution. Dans nos travaux, nous nous sommes intéressés aux méthodes statistiques.

La génération aléatoire uniforme est utilisée pour sélectionner des données à partir du domaine des entrées. Ce type d'approche, généralement appelé test statistique est étudié depuis longtemps. D'abord sur des données numériques [48, 49], cette approche s'est révélée avoir un pouvoir de détection inégal, devenant même peu convaincante lorsqu'elle était appliquée sur des programmes de complexité

[1] Étude menée auprès de plus de 200 acteurs (banques et industriels) pendant 18 mois.

[2] Dans ce cas, le terme de test de robustesse est généralement employé.

réaliste [19, 111]. Mais le tirage uniforme ne permet pas de détecter toutes les erreurs : il existe des cas particuliers correspondant à des sous-domaines d'entrée de probabilité faible. Cependant, elle permet de faire du test intensif et facilement automatisable.

Contrairement aux approches classiques qui se basent sur une distribution sur les entrées, nous nous sommes intéressés aux distributions sur les chemins d'exécution ou des traces du système à tester. Notre approche est donc mixte. Elle combine le test statistique avec le test structurel ou le test fonctionnel.

Test de logiciel et structures combinatoires

Motivation Récemment, l'étude et la simulation des processus stochastiques a largement profité des progrès dans le domaine de la génération aléatoire de structures combinatoires. Les travaux précurseurs de Wilf et Nijenhuis [93, 119] ont amené à l'élaboration d'algorithmes efficaces pour la génération aléatoire et uniforme d'une variété de structures combinatoires. En 1994, Flajolet, Zimmermann et Van Cutsem [56] ont généralisé et systématisé cette approche en s'appuyant sur une décomposition récursive non ambiguë des structures combinatoires à engendrer. Leurs travaux constituent la base d'un outil puissant de génération uniforme d'objets complexes comme des graphes, des arbres, des mots, des chemins...

Dans cette thèse, nous explorons l'usage de tels concepts et outils pour le test statistique de logiciel.

Contribution Notre approche consiste à utiliser les outils de génération de structures combinatoires dès qu'il existe une description sous forme de graphe des comportements du système à tester. Cette description peut être :

- le graphe de contrôle ou de flots de données
- une représentation graphique du système comme un réseau de Pétri, un Statechart...
- un système de transitions, un automate, une machine d'états, une structure de Kripke, etc.

Nous verrons que toutes ces descriptions peuvent être décrites sous forme d'une structure combinatoire étiquetée.

La génération uniforme peut être utilisée pour sélectionner des chemins à partir de l'ensemble des chemins d'exécution ou des traces du système à tester, ou parmi un sous-ensemble satisfaisant certaines conditions de couverture, ces ensembles étant tous modélisables par des structures combinatoires. La génération aléatoire uniforme de chemins permet ainsi de mieux couvrir les cas peu probables que la génération aléatoire uniforme d'entrées couvrait avec difficulté.

Notre approche a été appliquée au test statistique structurel. Cette application a permis le développement d'un premier prototype appelé AuGuSTe. Nous avons réalisé des expériences à l'aide de mutants afin d'évaluer expérimentalement notre approche et de la comparer à la méthode de test statistique structurel définie par Pascale Thévenod-Fosse et Hélène Waeselynck [110, 111].

Ces travaux ont déjà donné lieu à une publication [61] dans le *workshop* français AFADL (Approches Formelles dans l'Assistance au Développement de Logiciels) en 2000 et à une publication [63] dans la conférence IEEE ASE (*Automated Software Engeneering*) en 2001. Une présentation de notre outil AuGuSTe [62] est programmé pour le prochain *workshop* français AFADL qui aura lieu en juin 2004.

Structure de la thèse

Ce mémoire est divisé en trois parties : la première partie est consacrée à l'introduction des méthodes de tests dynamiques ainsi qu'aux recherches qui s'effectuent actuellement dans ce domaine, la deuxième partie est consacrée à la présentation de notre méthode et la troisième présente une première application de notre méthode, un prototype et les expériences effectuées.

Le **chapitre 1** est une introduction au test dynamique. Nous y présentons les trois grandes méthodes de sélection que sont le test statistique, le test structurel et le test fonctionnel.

Le **chapitre 2** présente un échantillon de méthodes dynamiques utilisant la sélection de chemins et la résolution de contraintes.

Le **chapitre 3** est une introduction aux structures combinatoires. Nous y présentons également des techniques de génération aléatoire de tels objets et les outils actuellement disponibles.

Le **chapitre 4** présente notre approche du test statistique basée sur un tirage aléatoire de chemins dans un graphe.

Le **chapitre 5** illustre l'application de notre approche au test statistique structurel.

Le **chapitre 6** présente le prototype AuGuSTe développé dans le but d'évaluer notre méthode.

Le **chapitre 7** est une présentation des expériences et des résultats obtenus.

Ce mémoire se termine par un bilan de nos travaux et propose quelques perspectives de recherche.

Première partie

Le test de logiciel

Chapitre 1

Introduction au test dynamique

Il existe de nombreuses méthodes de tests. Ces méthodes se répartissent en deux familles : les méthodes de "test statique" et les méthodes de "test dynamique" :

- Le test statique est appliqué sur une description du programme ou directement sur le texte du programme mais sans exécuter ce dernier. Il consiste en une analyse du texte dans le but d'en déduire des propriétés : on peut par exemple, détecter des variables utilisées sans être au préalable initialisées, des morceaux du programme qui ne seront jamais exécutés (c'est ce que l'on appelle du *code mort*), des boucles sans fin, etc. Il existe différents procédés de test statique dont les plus connus sont l'analyse d'anomalies (ex : typage impropre, portion de code isolées, etc.), l'évaluation symbolique, l'interprétation abstraite et l'inspection [58, 3].
- Le test dynamique est la technique de test la plus répandue. Elle repose sur l'exécution du programme sur un sous-ensemble des données et à la vérification que le résultat attendu est satisfaisant. Les domaines des entrées d'un programme étant généralement très grands, il n'est pas réaliste d'effectuer du test *exhaustif* c'est-à-dire une exécution sur toutes les entrées possibles. Il faut donc sélectionner un sous-ensemble pertinent, c'est-à-dire un sous-ensemble de données le plus efficace pour trouver des erreurs dans le programme.

Dans ce mémoire, nous nous intéressons à la famille des méthodes de test dynamique.

Le test dynamique se décompose en quatre étapes [58] :

1. la **sélection** qui consiste à choisir judicieusement un sous-ensemble d'entrées, que l'on appelle *jeu de tests*, parmi toutes les entrées possibles.
2. l'**exécution** qui consiste à faire exécuter le programme sur les entrées choisies à l'étape précédente.
3. l'**analyse** qui consiste à déterminer si un test a échoué ou non. Pour cela,

on s'intéresse aux sorties du programme et on vérifie si elles sont conformes ou non aux résultats attendus, déterminés par un *oracle*[1].

Si l'objectif du test est de vérifier un système alors on considère que le test a échoué dès qu'une erreur est révélée, par contre, si son objectif est de mettre en évidence un maximum d'erreurs, le test est un échec si aucune erreur n'est révélée.

4. l'**évaluation**[2] de la qualité des tests effectués qui permet entre autres de décider ou non de l'*arrêt du test*.

L'ensemble des méthodes de cette famille se classe en fonction de leur technique de sélection des données d'entrées. Pour sélectionner les tests, le testeur utilise les documents mis sa disposition lors de la phase de test c'est-à-dire les spécifications, les caractéristiques des entrées et les textes du programme. Si la sélection se fait à partir d'une distribution, par exemple sur le domaine des entrées, on parle de test *statistique* (voir section 1.1), si elle se fait à partir des textes du programme, on parle de test *structurel* (voir section 1.2), enfin si elle se fait à partir des spécifications, on parle de test *fonctionnel* (voir section 1.3). Afin d'augmenter l'efficacité d'un jeu de test, on peut être amené à combiner ces trois types de sélection : la section 1.4 présente les premiers travaux sur ce sujet.

1.1 L'approche *statistique* ou *aléatoire*

Le principe du test statistique est de tirer des données dans le domaine d'entrée selon une certaine distribution. Dans sa version classique, la distribution est uniforme : on ne tient compte ni de la spécification ni du code du programme testé. La distribution des données d'entrée peut également être estimée en fonction de l'exploitation future du système à tester. On parle alors de test aléatoire opérationnel [50, 111].

Le test aléatoire uniforme permet d'assurer une bonne couverture des programmes dont les chemins d'exécution[3] ont des domaines d'entrée de tailles (et donc de probabilités) comparables, mais dans le cas contraire, il doit être complété par des tests aux limites. En effet, il existe des cas particuliers (ex : les cas d'exception) qui correspondent à des sous-domaines d'entrée de probabilité trop faible.

L'avantage des méthodes de test basées sur le test statistique est qu'à partir du moment où cette méthode est automatisée, elle permet de faire du **test intensif**. Lorsque que la distribution est uniforme et le domaine d'entrée est composé

[1] Le problème de l'oracle ne sera pas abordé dans ce mémoire.

[2] L'étape d'évaluation n'est pas nécessairement explicite.

[3] Intuitivement, un chemin d'exécution est une mise en séquence possible des actions du programme.

d'intervalles numériques, il est facile d'automatiser ce type de test, mais ce n'est malheureusement pas toujours le cas.

Le test aléatoire uniforme est aussi utilisé pour "évaluer" expérimentalement toute nouvelle méthode de test : en effet, il serait souhaitable que la méthode de test élaborée fasse au moins aussi bien qu'une simple sélection aléatoire uniforme de données d'entrée. Les pouvoirs de détection de la nouvelle méthode est comparé avec celui de la sélection uniforme sur des ensembles de "mutants" (cf. section 7.1) de programmes, c'est-à-dire des programmes où des fautes ont été injectés [40].

Exemple 1 (Test aléatoire)
Dans le cas du programme erroné `abs` *ci-dessous, il s'agit de tirer de manière aléatoire des entiers pour* x*. La probabilité d'obtenir un* $x = 0$*, et de découvrir l'erreur, est très faible.*

E : `int abs (x:int)`
$P1$: `if x>0`
$B1$: `then return(x) ;`
`else`
$P2$: `if x=0`
$B2$: `then return(1) ;`
$B3$: `else return(-x) ;`

1.2 L'approche *boîte de verre* ou *test structurel*

L'objectif du test structurel est de tester **comment le programme fait ce qu'il a à faire**. La sélection des données d'entrée se fait à l'aide d'une description de la structure du programme qui peut être le graphe de contrôle ou le flot de données [58].

En 1996, Ferguson et Korel [55] mentionnent différents types d'approches pour générer des données de test :

- l'approche "orientée chemin" (*path-oriented*) qui consiste à sélectionner un chemin du programme qui atteint l'élément choisi (instruction, enchaînement, etc.) puis de calculer les entrées qui permettent l'exécution de ce chemin.
- l'approche "orientée but" (*goal-oriented*) qui consiste à générer des données d'entrées qui permettent d'atteindre l'élément choisi (instruction, enchaînement, etc.) sans s'occuper du chemin emprunté. L'étape de sélection des chemins est inexistante.

Dans ce mémoire, nous nous intéressons aux méthodes "orientées chemin". Dans ce cadre, le test structurel est donc basé sur l'exécution d'un sous-ensemble

fini des chemins du programme en sélectionnant les données d'entrée qui permettent de les déclencher. La présence de boucles, dont la longueur dépend des données d'entrée, dans un programme peut impliquer un ensemble potentiellement infini de chemins, c'est pourquoi on ne s'intéresse qu'à un sous-ensemble judicieusement sélectionné. La sélection se fait à l'aide de **critères de couverture**. C'est également le critère de couverture qui détermine le nombre de tests à effectuer.

Il existe de nombreux critères de couverture [74], nous présenterons en section 1.2.1 les plus fréquemment utilisés pour la couverture structurelle de graphe de contrôle. En section 1.2.2 puis 1.2.3, nous expliquons comment à partir des chemins sélectionnés, on peut obtenir des données d'entrée qui assurent leur exécution.

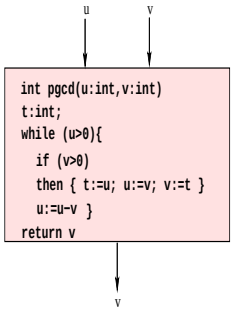


Fig. 1.1 : *Test structurel= tester ce que fait chaque élément du programme*

1.2.1 Graphe de contrôle et critères de couverture

Il existe plusieurs façons d'abstraire un programme, parmi lesquelles on trouve le **graphe de contrôle**. Il s'agit d'une représentation simplifiée, sans appel de fonction ou gestion d'exception, de l'algorithme d'un programme. Il est construit directement à partir du code source de ce programme. C'est un graphe orienté fini dont :

- les sommets sont les blocs d'instructions indivisibles maximaux et les prédicats qui apparaissent dans les instructions conditionnelles ou les boucles,
- les arcs correspondent aux transferts de contrôle possibles entre ces sommets.

Un **bloc d'instructions indivisible maximal** est une portion linéaire maximale de code qu'on exécute toujours du début jusqu'à la fin. Ce bloc est formé d'une suite d'instructions élémentaires telle que le contrôle passe consécutivement de l'une à l'autre sans choix possible (pas de *While*, *IfThenElse*, *IfThen*, etc.).

Les instructions conditionnelles *IfThenElse* seront représentées par trois sommets et deux arcs, les instructions *IfThen* par deux sommets et deux arcs et les boucles conduiront à un circuit dans le graphe.

Dans la suite, on distingue les blocs d'instructions indivisibles maximaux par des sommets *carrés* et les prédicats par des sommets *ronds*.

Un graphe de contrôle ne possède qu'un seul point d'entrée et un seul point de sortie. Si d'origine, il possède plusieurs sorties, par exemple des instructions `return` explicites, alors on rajoute un bloc de sortie fictif que l'on note généralement S et on relie toutes les sorties réelles à ce bloc de sortie. De plus, tous les sommets sont atteignables à partir du point d'entrée et mènent au point de sortie.

Exemple 2 (Un programme et son graphe de contrôle : `tordu`)

```
E : procedure tordu (t: array of int, borne: int){
    i,j: int;
B1 :  i:=j; i:=0;
P1 :  while(i<borne){
P2 :   if (t[i] mod 2 == 0)
B2 :    then write(t[i]);
B3 :   read(j); t[i]:=t[j]+1};
B4 :  j:=2×j;
P3 :  if(j>0)
B5 :   then { i:= 0; write (``possible'') }
B6 :   else write (``impossible'')
}
```

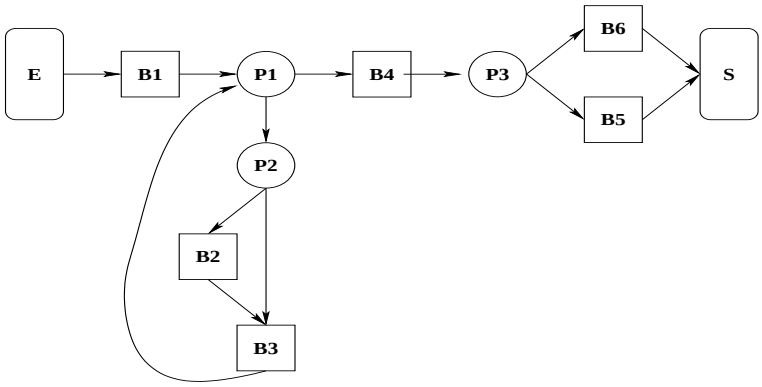


Fig. 1.2 : *Graphe de contrôle du programme* `tordu`

Définition 1

Un **chemin complet** *dans un graphe de contrôle est un chemin du graphe (suite de sommets ou d'arcs) qui va du point d'entrée du graphe à son point de sortie.*

Un **chemin élémentaire** *est un chemin complet du graphe tel que chaque arc n'est emprunté au plus qu'une seule fois.*

La **longueur d'un chemin** *est définie par le nombre d'arcs (resp. sommets) qui le composent.*

Dans la suite, on ne s'intéresse qu'à des chemins complets, puisque ce sont les seuls qui modélisent des exécutions du programme à tester.

Exemple 3

Le graphe de contrôle de la figure 1.2 a 6 chemins élémentaires qui vont du sommet entrant E au sommet sortant S :

- $E.B1.P1.B4.P3.B5.S$
- $E.B1.P1.B4.P3.B6.S$
- $E.B1.P1.P2.B2.B3.P1.B4.P3.B5.S$
- $E.B1.P1.P2.B2.B3.P1.B4.P3.B6.S$
- $E.B1.P1.P2.B3.P1.B4.P3.B5.S$
- $E.B1.P1.P2.B3.P1.B4.P3.B6.S$

Selon la méthode de tests utilisée, on peut annoter le graphe de contrôle par des informations pertinentes, comme par exemple les conditions de branchement.

Les méthodes de test basées sur le graphe de contrôle ont pour principe de sélectionner des entrées telles que l'exécution du programme à partir de ces entrées permet de couvrir un ensemble d'éléments du graphe de contrôle.

Les principaux critères de couverture structurelle qui se basent sur le graphe de contrôle sont le critère "tous les chemins", le critère "toutes les instructions", c'est-à-dire les sommets carrés du graphe de contrôle, et le critère "tous les enchaînements".

Mais il en existe beaucoup d'autres [74] comme la couverture de tous les noeuds de décision, c'est-à-dire les sommets ronds, ou encore la couverture dits des conditions multiples qui consistent à couvrir toutes les combinaisons possibles de noeuds de décisions.

1.2.1.1 Le critère "tous les chemins"

Ce critère impose de tester au moins une fois tous les chemins exécutables entre le noeud d'entrée du graphe de contrôle et tous les noeuds de sortie. Ce critère, à priori simple, devient très vite trop coûteux et surtout impraticable dès lors qu'un programme contient une boucle qui implique l'existence d'une infinité de

chemins à tester. Dans le cas général, il est donc utopique de vouloir satisfaire ce critère sans y ajouter de restrictions. Nous présentons ci-dessous deux restrictions possibles :

1. tous les chemins limite-intérieure (ou i-chemins) [117] : demande d'itérer chaque boucle au plus une fois, puis dans un deuxième essai au plus deux fois, puis dans un i-ième essai au plus i fois. Cette restriction permet ainsi de réduire le nombre de chemins pendant le test.
2. tous les chemins de taille bornée : on se limite à ne tester que les chemins de longueur inférieure ou égale à n.

1.2.1.2 Le critère "toutes les instructions"

Ce critère exige d'exécuter au moins une fois chaque instruction : ce qui revient à passer au moins une fois par chaque sommet du graphe de contrôle.

Il est relativement facile à mettre en oeuvre car il existe toujours au moins un sous-ensemble fini de chemins qui le satisfait, et ce, que le programme contienne des boucles ou non. Il s'est avéré cependant nettement insuffisant expérimentalement [78].

Exemple 4 (Un programme erroné du calcul de factorielle)
Spécification : la fonction factorielle `fact` *prend un entier i en paramètre.*
- *Si i est négatif alors la fonction retourne 0.*
- *Si i est nul alors la fonction retourne 1.*
- *Si i est positif alors la fonction retourne $!i$.*

Programme erroné :

```
E :int fact(i:int)
P1 : if (i≥0)
     then
P2 :  if (i=0)
B1 :   then return(1) ;
B2 :   else return(i×fact(i-1)) ;
```

Dans l'exemple 4, le critère impose de passer par les instructions suivantes au moins une fois : $return(1)$ et $return(i \times fact(i-1))$. En prenant comme entrée un nombre i strictement positif, on peut couvrir le critère "toutes les instructions" *avec un seul test*, par exemple : $i = 1$. Au premier appel de `fact`, on passe par $B2$ et avec l'appel récursif, on passe par $B1$. Si l'on se base sur la spécification de factorielle donnée, ce test, même s'il couvre le critère, ne permet pas typiquement de révéler l'erreur glissée dans le programme.

Le critère "toutes les instructions" fait partie des critères minimaux requis mais n'est absolument pas suffisant.

1.2.1.3 Le critère "tous les enchaînements"

Ce critère exige d'emprunter au moins une fois chaque enchaînement possible d'instructions : ce qui revient à passer au moins une fois par chaque arc du graphe de contrôle.

C'est le critère le plus souvent utilisé [78] en pratique car il est moins coûteux que "tous les chemins" et il assure une meilleure détection que "toutes les instructions".

Dans l'exemple 4, le critère "tous les enchaînements" impose de couvrir les enchaînements suivants :

- passer dans la branche `Then` de $P1$ puis par la branche `Then` de $P2$
- passer dans la branche `Then` de $P1$ puis par la branche `Else` de $P2$
- passer dans la branche `Else` de $P1$

Pour couvrir ce critère, il nous faut au moins un entier strictement positif, par exemple 1, pour couvrir les deux premiers cas et un entier strictement négatif, par exemple -1, pour couvrir le dernier cas. L'erreur est alors découverte.

1.2.2 Des chemins aux prédicats

Étant donné un ensemble de chemins qui satisfait un des critères de couverture, il faut maintenant trouver l'ensemble des entrées qui permettent de passer par ces chemins. Pour cela, on calcule le prédicat associé à chacun des chemins.

On peut considérer un chemin du programme comme une suite dont les éléments sont soit :

- une affectation
- un prédicat de conditionnelle ou de boucle (correspondant à l' exécution de la branche *Then* ou du corps de la boucle)
- la négation d'un prédicat de conditionnelle ou de boucle (correspondant à l'exécution de la branche *Else*, la sortie d'une conditionnelle incomplète ou d'une boucle)

Le **prédicat d'un chemin** est une condition sur les données d'entrée qui provoque l'exécution de ce chemin. Classiquement, la construction du prédicat de chemin se fait de la manière suivante :

- On initialise la construction en considérant que toute variable x a une valeur courante initiale notée x_0
- toute affectation $x := expr$ sera transformée en une égalité $x_{i+1} = \widetilde{expr}$ où i est l'indice de la valeur courante de x et $\widetilde{expr}$ est l'expression $expr$ où tout variable a été instanciée par sa valeur courante. La valeur courante de x devient x_{i+1}
- tout prédicat P (resp. négation de prédicat *non* P) d'une conditionnelle ou d'une boucle est remplacé par le prédicat $\tilde{P}$ (resp. $\widetilde{\neg P}$) qui est le prédicat

P (resp. *non* P) où toutes les variables ont été remplacées par leur valeur courante.

Le prédicat final est la conjonction de toutes ces formules.

Exemple 5

E :`int pgcd(u:int,v:int)`
`i,t: int;`
$P1$: `while (u>0){`
$P2$: `if (v>u)`
$B1$: `then{ t:=u; u:=v; v:=t};`
$B2$: `u:=u-v;}`
S : `return(v)`

Dans le programme `pgcd` *ci-dessus, si l'on exprime le prédicat du chemin* $E.P1.S$ *c'est-à-dire du chemin ne passant pas par la boucle, cela donne :*

$$\texttt{predicat}(u_0,v_0,E.P1.S)\ :=\neg(u_0>0)$$

Si l'on exprime le prédicat du chemin passant deux fois par la boucle, et une fois dans la branche `Then` *(par exemple au premier passage dans la boucle), cela donne :*

$\texttt{predicat}(u_0,v_0,E.P1.P2.B1.B2.P1.P2.B2.P1.S)\ :=$

$$(u_0>0)\texttt{and}(v_0>u_0)\texttt{and}(t_0=u_0)\texttt{and}(u_1=v_0)\texttt{and}(v_1=t_0)\texttt{and}(u_2=u_1-v_1)$$
$$\texttt{and}(u_2>0)\texttt{and}(\neg(v_1>u_2))\texttt{and}(u_3=u_2-v_1)\texttt{and}(\neg(u_3>0))$$

où :

- $u_0>0$ *correspond au premier passage dans le corps de la boucle*
- $v_0>u_0$ *correspond au passage dans la branche* `Then`
- $u_2>0$ *correspond au second passage dans le corps de la boucle*
- $\neg(v_1>u_2)$ *correspond au non passage dans la branche* `Then`
- $\neg(u_3>0)$ *correspond à la sortie de la boucle*

Ce prédicat peut s'exprimer également qu'avec les variables d'entrée :

$\texttt{predicat}(u_0,v_0,E.P1.P2.B1.B2.P1.P2.B2.P1.S)\ :=$

$$(u_0>0)\texttt{and}(v_0>u_0)\texttt{and}(v_0-u_0>0)$$
$$\texttt{and}(\neg(u_0>v_0-u_0))\texttt{and}(\neg(v_0-u_0-u_0>0))$$

qui se simplifie en :

$\texttt{predicat}(u_0,v_0,E.P1.P2.B1.B2.P1.P2.B2.P1.S)\ :=\ (u_0>0)\texttt{and}(v_0=2u_0)$

Une fois, l'ensemble des prédicats construit, il faut résoudre chacun de ces prédicats afin de récupérer des données d'entrée qui le satisfasse et donc provoquent un passage par le chemin auquel il est associé : ce sont les tests proprement dits. Le principe de la résolution de prédicats est expliqué dans la section 1.2.3.

Il n'est pas toujours possible de résoudre les prédicats : si un prédicat n'a pas de solution, c'est que le chemin associé est infaisable. L'existence de chemins infaisables complique l'application de toutes les méthodes de test.

L'identification des chemins infaisables est un problème qui dépend de la nature des prédicats associés. Si le prédicat n'est pas linéaire alors le problème qui consiste à déterminer s'il existe une solution est indécidable [37, 118]. Par contre, si le prédicat est linéaire selon les variables d'entrées, le problème de faisabilité devient décidable : il existe des techniques de programmation linéaire qui permettent de trouver une solution à de telles contraintes [118].

1.2.3 Résolution des prédicats

1.2.3.1 Rappels de définitions

Une **contrainte** est une propriété entre différentes inconnues (les variables) qui doit être vérifiée. Chaque variable prend ses valeurs dans un ensemble donné (fini ou non) que l'on appelle domaine. Selon le domaine considéré, il peut être décrit soit sous la forme d'un ensemble de valeurs soit par un intervalle ou une union d'intervalles. Dans ce mémoire, nous ne considérons que les domaines finis (*Finite Domain*). Ainsi, une contrainte peut être vue comme une restriction des valeurs que peuvent prendre simultanément les variables. Il existe différentes caractérisations d'une contrainte (linéaire, relationnelles, booléennes, etc.), pour plus d'informations, nous renvoyons le lecteur à [116, 107].

Un problème de satisfaction de contraintes CSP (*Constraint Solving Problem*) est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables.

Définition 2 (CSP [84])
Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *tel que :*
- $\mathcal{X} = \{x_1, ..., x_n\}$ *est un ensemble fini de variables*
- $\mathcal{D} = \{D_{x_1}, ..., D_{x_n}\}$ *est un ensemble fini de domaines où* D_{x_i} *est le domaine contenant toutes les valeurs possibles de la variables* x_i*.*
- $\mathcal{C} = \{c_1, ..., c_m\}$ *est un ensemble fini de contraintes où chaque contrainte* c_i *porte sur un sous-ensemble* $\{x_{i_1}, ..., x_{i_k}\}$ *de* $\mathcal{X}$*.*

Exemple 6 (CSP) *Soit le CSP*$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *suivant :*
- $\mathcal{X} = \{a, b, c, d\}$
- $\mathcal{D} = \{D_a, D_b, D_c, D_d\}$ *avec*
  - $D_a = D_b = D_d = \{1, 2, 3, 4, 5\}$
  - $D_c = \{0, 1\}$
- $\mathcal{C} = \{c_1, c_2, c_3\}$ *avec*
  - $c_1 : \ a \neq b$

- $c_2 : c \neq d$
- $c_3 : a + c < b$

Ce CSP comporte 4 variables a, b, c et d, chacune pouvant prendre 5 valeurs $\{1, 2, 3, 4, 5\}$ *sauf c qui ne peut en prendre que 2* $\{0, 1\}$*. Ces variables doivent respecter les contraintes suivantes : a et b doivent être différents, c doit être différent de d et la somme de a et b doit être strictement inférieur à b.*

Définition 3 (affectation, partielle ou totale)
On appelle **affectation** *le fait d'instancier certaines variables par des valeurs de leur domaine. On notera* $A = \{(x_1, v_1), (x_2, v_2), ..., (x_n, v_n)\}$ *l'affectation qui instancie la variable* x_1 *par la valeur* v_1*, la variable* x_2 *par la valeur* v_2*, ..., la variable* x_n *par la valeur* v_n*.*

Une affectation est dite **totale** *si elle instancie toutes les variables du problème ; elle est dite* **partielle** *si elle n'en instancie qu'une partie.*

*On parle aussi d'***instance** *(label) lorsqu'elle ne concerne qu'une seule variable.*

Exemple 7 *Sur le CSP précédent :*

- $A = \{(a, 1), (b, 1), (c, 1)\}$ *est une affectation partielle*
- $A = \{(a, 1), (b, 1), (c, 1), (d, 1)\}$ *est une affectation totale*

Définition 4 (satisfaire ou violer une contrainte)
Une affectation A **viole** *une contrainte c si toutes les variables de c sont instanciées dans A, et si la relation définie par c n'est pas vérifiée pour les valeurs des variables de c définies dans A.*

Dans le cas contraire, on dit que l'affection A **satisfait** *la contrainte c.*

Exemple 8 *Sur le CSP précédent :*

- *l'affectation partielle* $A = \{(a, 1), (b, 1)\}$ *viole la contrainte* $c_1 : a \neq b$ *mais pas les contraintes* c_2 *et* c_3 *dans la mesure où certaines de leurs variables ne sont pas instanciées dans A*
- *l'affectation totale* $A = \{(a, 1), (b, 1), (c, 1), (d, 0)\}$ *satisfait la contrainte* $c_2 : c \neq d$ *mais viole les contraintes* c_1 *et* c_3

Définition 5 (affectation consistante et inconsistante)
Une affectation est **consistante** *si elle ne viole aucune contrainte, et* **inconsistante** *si elle viole une ou plusieurs contraintes.*

Exemple 9 *Sur le CSP précédent :*

- *l'affectation partielle* $A = \{(c, 1), (d, 0)\}$ *est consistante car elle ne viole aucune contrainte*
- *l'affectation totale* $A = \{(c, 1), (d, 0), (a, 3), (b, 5)\}$ *est consistante car elle ne viole aucune contrainte*

- *l'affectation partielle* $A = \{(a,1),(b,1)\}$ *est inconsistante car elle viole la contrainte* c_1
- *l'affectation totale* $A = \{(a,1),(b,1),(c,1),(d,0)\}$ *est inconsistante car elle viole les contraintes* c_1 *et* c_3

Définition 6 (solution d'un CSP)
Une **solution** *est une affectation totale consistante, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.*

Exemple 10 (Un CSP et une de ses solutions)
Sur le CSP précédent, l'affectation totale $A = \{(c,1),(d,0),(a,3),(b,5)$ *est consistante, il s'agit donc d'une solution du CSP.*

1.2.3.2 Résolution

Résoudre un prédicat de chemin consiste à trouver une solution au CSP associé.

Exemple 11 (Prédicat et CSP associé)
Si l'on reprend le prédicat de l'exemple 5 :
`predicat(`u_0`,`v_0`,`$E.P1.P2.B1.B2.P1.P2.B2.P1.S$`) :=`

$$(u_0 > 0)\texttt{and}(v_0 > u_0)\texttt{and}(t_0 = u_0)\texttt{and}(u_1 = v_0)\texttt{and}(v_1 = t_0)\texttt{and}(u_2 = u_1 - v_1)\texttt{and}(u_2 > 0)\texttt{and}(\neg(v_1 > u_2))\texttt{and}(u_3 = u_2 - v_1)\texttt{and}(\neg(u_3 > 0))$$

Le CSP$(\mathcal{X},\mathcal{D},\mathcal{C})$ *associé est :*
- $\mathcal{X} = \{u_0, v_0, t_0, u_1, v_1, u_2, u_3\}$
- $\mathcal{D} = \{D_{u_0}, D_{v_0}, D_{t_0}, D_{u_1}, D_{v_1}, D_{u_2}, D_{u_3}\}$ *avec* $\forall D_i \in \mathcal{D},\ D_i = I\!N$
- $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$ *avec*
  - $c_1 :\ u_0 > 0$
  - $c_2 :\ v_0 > u_0$
  - $c_3 :\ t_0 = u_0$
  - $c_4 :\ u_1 = v_0$
  - $c_5 :\ v_1 = t_0$
  - $c_6 :\ u_2 = u_1 - v_1$
  - $c_7 :\ u_2 > 0$
  - $c_8 :\ \neg(v_1 > u_2)$
  - $c_9 :\ u_3 = u_2 - v_1$
  - $c_{10} :\ \neg(u_3 > 0)$

Les techniques de résolution des CSP sont basées sur les techniques de consistance c'est-à-dire une propagation des valeurs avec réduction des domaines de variables. En générale, ce sont des combinaisons des trois principales méthodes de résolution [116] :

- la simplification des problèmes (*problem reduction*) qui consiste à transformer le CSP en un problème plus facile à résoudre ou que l'on sait insatisfiable
- la recherche de solutions (*search*) qui consiste à choisir une variable, puis une valeur pour cette variable (cette phase est appelée *labeling*) qui soit compatible avec les contraintes et les autres instances déjà choisies. Si toutes les variables sont instanciées alors le problème est résolu, sinon les assignations sont remises en cause en commençant par la dernière (cette phase est appelée *backtracking*). Au final, soit une solution est trouvée soit toutes les combinaisons d'instances ont été essayées et ont échoué.
- la synthèse des solutions possibles (*solution synthesis*) est un algorithme de recherche qui construit étape par étape plusieurs affectations partielles simultanément. Il peut être vu comme une simplification du problème dans lequel une contrainte qui porte sur toutes les variables est créée et réduite à un ensemble qui contient toutes les solutions et uniquement celles-là. Cette méthode est généralement utilisée lorsque l'on recherche toutes les solutions d'un problème.

Pour plus de précisions sur ces approches, nous renvoyons le lecteur au livre de de Edward Tsang [116] et/ou en annexe C.

1.2.3.3 Le cas particulier des tableaux

Lorsque le prédicat à résoudre contient au moins une variable de type tableau, le procédé de résolution se complique : le problème des tableaux apparaît lorsque la valeur d'un indice est inconnue au moment de la résolution. Nous présentons dans cette section la méthode de résolution de prédicats contenant des tableaux proposée dans [101].

Le principe de cette méthode consiste à retarder les substitutions jusqu'à ce que l'on connaisse suffisamment d'informations sur l'indice, en créant de nouvelles instances du tableau. Pour chaque nouvelle instance, l'élément d'indice i du tableau à la même valeur que l'élément d'indice i de l'instance précédente du tableau sauf pour l'élément à qui on vient d'assigner une nouvelle valeur. Ainsi par exemple, si l'instance courante est k alors après l'affectation

$$T[i] := n$$

on obtient une nouvelle instance $k+1$ du tableau telle que :

$$T_{k+1}[j] = T_k[j], \forall j \neq i$$
$$T_{k+1}[i] = n$$

1.2.3.4 Les solveurs de contraintes

Il existe de nombreuses manières d'optimiser la résolution d'un problème CSP. En général, les algorithmes existants s'appliquent à des CSP bien particuliers et permettent d'obtenir une, plusieurs ou une des meilleurs solutions pour ce problème. Par conséquence, à chaque type de problème, ses heuristiques et donc son solveur de contraintes le plus efficace.

Les solveurs de contraintes sont généralement construits à partir de librairies de domaines finis et des mécanismes de base provenant d'environnement de programmation par contraintes comme ECL^{i}PS^e [51] ou SICStus Prolog.

Les variables booléennes prennent leurs valeurs dans le domaine booléen ou le domaine d'entier réduit à l'ensemble $\{0, 1\}$ et les contraintes booléennes se traduisent en opérations booléennes qui seront traitées à l'aide d'unifications et de simplifications symboliques. Par exemple, la contrainte booléenne $x \wedge y$ se traduit par l'opération booléenne `and(x,y,R)` où R est le résultat de $x \wedge y$.

Les variables numériques prennent leurs valeurs dans des unions d'intervalles (limités par les valeurs maximales et minimales représentables par une machine), et les contraintes numériques se traduisent par des opérations arithmétiques et de comparaison qui seront traitées à l'aide d'unifications et de réductions d'intervalles des domaines. Par exemple, la contrainte booléenne $x < y$ se traduit par l'opération booléenne `inf(x,y,R)` où R est le résultat de $x < y$.

Le principe d'un solveur de contraintes peut se résumer en quatre étapes principales :

1. simplifier le problème donné, puis stocker les contraintes non encore satisfaites dans un **magasin de contraintes** (*constraint-store*) jusqu'à l'affectation de variables qui pourrait la faire de nouveau progresser.
2. choisir puis instancier une variable selon la ou les heuristiques implémentées
3. L'affectation d'une variable "réveille" la propagation des contraintes concernées, qui peuvent disparaître (lorsqu'elles sont résolues), ou réveiller/créer d'autres contraintes, ou amener à un échec de la résolution (domaine vide ou échec de lors de l'unification).
4. – si toutes les variables sont instanciées alors s'arrêter sur un succès
   – si il y a un échec alors opérer un *backtracking*
   – si il y a un échec et aucun *backtracking* n'est possible alors s'arrêter sur un échec
   – sinon retourner à l'étape 2.

L'efficacité d'un solveur de contraintes repose énormément sur le choix de la variable à instancier durant les étapes de *labelling*. Les heuristiques de choix classiques sont basées sur le réveil d'un nombre maximal de contraintes et sur le plus petit domaine.

Nous présentons maintenant une stratégie de résolution basée sur la randomisation qui permet une résolution rapide tout en assurant que la première solution retournée ne soit pas systématiquement la même d'une résolution à l'autre.

Cette stratégie est issue du solveur de contraintes, implémenté dans ECL^{i}PS^e, de l'outil GATeL, présenté au chapitre 2. Elle est basée sur une élimination successive de toutes les contraintes en utilisant une procédure d'instanciation non déterministe des variables de ces contraintes. Après l'instanciation d'une variable, toutes les contraintes dans lesquelles elle apparaissait sont propagées et un processus de réduction des domaines est activé. La propagation permet de vérifier à tout instant la satisfiabilité de l'ensemble des contraintes et a pour conséquence le réveil ou encore l'introduction de nouvelles contraintes. La variable instanciée est celle qui réveille le maximum de contraintes et dont le domaine de valeur est le plus petit. Le processus est itéré jusqu'à l'élimination de toutes les contraintes.

Exemple 12

Reprenons le problème CSP correspondant au prédicat de l'exemple 5 :

`predicat(`u_0`,`v_0`,`$E.P1.P2.B1.B2.P1.P2.B2.P1.S$`) :=`

`(`u_0`>0)and(`v_0`>`u_0`)and(`t_0`=`u_0`)and(`u_1`=`v_0`)and(`v_1`=`t_0`)and(`u_2`=`u_1`-`v_1`)`

`and(`u_2`>0)and(¬(`v_1`>`u_2`))and(`u_3`=`u_2`-`v_1`)and(¬(`u_3`>0))`

qui est :

- $\mathcal{X} = \{u_0, v_0, t_0, u_1, v_1, u_2, u_3\}$ *est un ensemble de variables* entières
- $\mathcal{D} = \{D_{u_0}, D_{v_0}, D_{t_0}, D_{u_1}, D_{v_1}, D_{u_2}, D_{u_3}\}$ *avec*
  - $D_{u_0} = D_{u_1} = D_{u_2} = D_{u_3} = [minInt..maxInt]$
  - $D_{v_0} = D_{v_1} = [minInt..maxInt]$
  - $D_{t_0} = [minInt..maxInt]$
- $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$ *avec*

$$\begin{cases} c_1 : u_0 > 0 \\ c_2 : v_0 > u_0 \\ c_3 : t_0 = u_0 \\ c_4 : u_1 = v_0 \\ c_5 : v_1 = t_0 \end{cases} \quad \begin{cases} c_6 : u_2 = u_1 - v_1 \\ c_7 : u_2 > 0 \\ c_8 : \neg(v_1 > u_2) \\ c_9 : u_3 = u_2 - v_1 \\ c_{10} : \neg(u_3 > 0) \end{cases}$$

L'ensemble des entiers ne pouvant être totalement représenté en machine, lors de l'utilisation d'un programme de résolution de contraintes, on est amené à réduire cet ensemble à l'intervalle $[minInt..maxInt]$ *où les valeurs* $minInt$ *et* $maxInt$ *représentent respectivement le plus petit et le plus grand entier représentables par la machine.*

Une phase de simplification permet de réduire les domaines et l'ensemble des contraintes à :

- $D_{u_0} = D_{t_0} = D_{u_2} = D_{v_1} = [1..\frac{maxInt}{2}]$
- $D_{u_1} = D_{v_0} = [2..maxInt]$
- $D_{u_3} = \{0\}$

- $\begin{cases} c_3 : t_0 = u_0 \\ c_{11} : u_0 = v_1 \\ c_{12} : u_0 = u_2 \\ c_{13} : v_0 = 2 \times u_0 \\ c_{14} : u_1 = 2 \times u_0 \end{cases}$

La première variable à instancier choisie par la stratégie de GATeL serait u_0 *car c'est la variable qui apparaît dans le plus de contraintes et qui a le plus petit domaine.*

Le solveur de contraintes que nous utilisons dans notre prototype (cf. chapitre 6) est issu du noyau de GATeL. La possibilité d'avoir une solution différente à chaque résolution d'un prédicat est très intéressant en test. Ainsi, si un chemin est sélectionné plusieurs fois, alors les entrées permettant son exécution ne seront pas forcément les mêmes à chaque fois.

1.3 L'approche *boîte noire* ou *test fonctionnel*

L'objectif du test fonctionnel consiste à tester **ce qu'est censé faire le programme** et non la façon dont il le fait. La structure du programme est donc inconnue ou ignorée et les données d'entrées sont sélectionnées en fonction des spécifications du programme. Les méthodes de test fonctionnel sont basées sur un découpage de domaine d'entrée de chaque variable en un ensemble fini de sous-domaines. Chaque sous-domaine d'une variable représente soit un ensemble de valeurs valides pour cette variable soit un ensemble de valeurs invalides.

Pour chaque variable de la spécification, un ensemble de valeurs est choisi tel qui permet de couvrir tous les sous-domaines y compris ceux qui représentent des valeurs invalides. Un test par ensemble de valeurs est alors sélectionné. Idéalement, l'ensemble de tests devrait couvrir toutes les combinaisons possibles de valeurs choisies pour chaque entrée mais le nombre de cas à considérer explose rapidement. Par ailleurs, la plupart des méthodes effectue également un test par valeurs aux limites (valeurs aux bornes). Une valeur aux limites est une valeur à la frontière de deux ou plusieurs sous-domaines. Tester les valeurs aux limites permet de détecter efficacement un grand nombre d'erreurs car les limites de domaines sont une cause fréquente d'erreur.

Le test fonctionnel inclut le **test de conformité** dont le but est de vérifier le comportement du système à tester par rapport à la spécification donnée à partir de son comportement et de ses interactions avec l'environnement [70, 114].

L'avantage de ce type de test est qu'il est indépendant d'une implémentation du système et donc réutilisable pour tout système implémentant la même spécification.

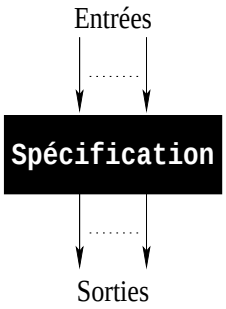


Fig. 1.3 : *Test fonctionnel*

Si la spécification est fournie en langage naturel ou est semi-formelle (spécification des méthodes UML [20] ou Merise [108]), l'utilisation de méthodes et/ou d'outils systématiques pour cette approche du test est délicate car ces spécifications contiennent souvent des ambiguïtés ou sont incomplètes. Pour faire du test boîte noire d'une manière systématique et rigoureuse, voire automatique, il faut disposer d'une spécification formelle.

Remarque 1 *De nombreux travaux [23, 47, 97, 82, 27] s'intéressent maintenant à la génération de tests directement à partir de spécifications UML en se basant sur une sémantique formelle.*

Les spécifications formelles permettent de raisonner sur les propriétés du système en "négligeant" les aspects algorithmiques et les choix de conception du système. Ce spécifications sont utilisables par des outils de preuve, d'évaluation symbolique ou de tests pour vérifier et valider le système.

Définition 7 (Spécification formelle [58])
Une spécification est dite **formelle** *si :*

- *elle est écrite en suivant une syntaxe bien définie*
- *la syntaxe est accompagnée d'une sémantique rigoureuse qui définit des modèles mathématiques correspondants aux réalisations acceptables de chaque spécification.*

Parmi les spécifications formelles les plus connues, on trouve les spécifications algébriques et les spécifications ensemblistes.

Les spécifications algébriques [18, 92] donnent une description axiomatique des fonctionnalités du système. Elles reposent sur la définition de types abstraits de données et des opérateurs (de création, de modification et d'observation) qui leurs sont associés. Ces opérateurs sont définis à partir de leur signature et de propriétés (axiomes).

L'ensemble des entrées de test est sélectionné de façon à couvrir les axiomes de la spécification [15]. Les tests consistent en des instances closes des axiomes que doit satisfaire le système. La sélection d'un nombre fini de valeurs des variables se fait par analyse de la spécification : les conditions apparaissant dans les axiomes permettent de dériver un nombre fini de domaines d'uniformité[4] ; une valeur pour chacun de ces domaines est choisie.

Les spécifications orientées modèle reposent sur la théorie des ensembles et sur la logique des prédicats : la modélisation du domaine est faite à l'aide d'objets ensemblistes (ensembles, relations, fonctions, etc.) et/ou d'invariants (contraintes). Les propriétés du système ne sont pas exprimées de façon explicite mais peuvent être déduites du modèle. Ces spécifications ensemblistes sont décrites à l'aide de langages formels comme VDM [73], Z [121] ou encore B [4].

L'ensemble des entrées de test est sélectionné de façon à couvrir les différents états du système. En réduisant sous forme normale disjonctive (DNF) les expressions définissant les opérations puis en faisant une analyse de cas sur la spécification, des domaines de test sont exhibées pour chaque opération [45]. De façon similaire l'état du système peut être partitionné en différents cas de test qui sont à la base de la construction d'un automate à états finis. On s'intéresse alors aux suites de tests couvrant toutes les transitions de l'automate.

D'autres approches des spécifications formelles sont basées sur des modèles graphiques. Les graphes utilisés sont généralement finis, orientés et étiquetés. L'interprétation associée aux arcs et noeuds de ces graphes dépend de l'utilisation qu'il en sera faite. Le graphe "de base" est le **système de transitions étiquetés** ou LTS (*Labelled Transition System*). Ce type de modèle est très utilisé pour la spécification et le test de protocoles de communication.

Définition 8 ([91])
Un système de transitions étiquetés est un quadruplet $(\mathcal{Q},q_0,\mathcal{A},\rightarrow)$ *où*
- $\mathcal{Q}$ *est un ensemble dénombrable non vide d'états,*
- $q_0 \in \mathcal{Q}$ *est l'état initial*
- $\mathcal{A}$ *est un ensemble dénombrables d'actions observables*
- $\rightarrow \subseteq \mathcal{A} \times (\mathcal{A} \cup \{\tau\}) \times \mathcal{A}$ *est la relation de transition où* $\tau \notin \mathcal{A}$ *représente les actions internes.*

Les étiquettes d'un LTS peuvent se distinguer entre elles selon ce que modélise ce LTS. Ainsi on peut être amené à différencier les entrées d'un système (transitions étiquetées `?`), ses sorties (transitions étiquetées `!`) et ses actions internes. Dans de tels cas, on a $\mathcal{A} = \mathcal{A}_I \bigcup \mathcal{A}_O \bigcup \mathcal{I}$ où $\mathcal{A}_I$ est l'ensemble des entrées possibles, $\mathcal{A}_O$ est l'ensemble des sorties possibles et $\mathcal{I}$ est l'ensemble des actions internes.

[4]Chacun de ces domaines repose sur l'hypothèse d'uniformité qui assure que si le résultat obtenu par une valeur issue d'un domaine est correct alors il en est de même pour tout résultat obtenu par toutes les autres valeurs de ce domaine.

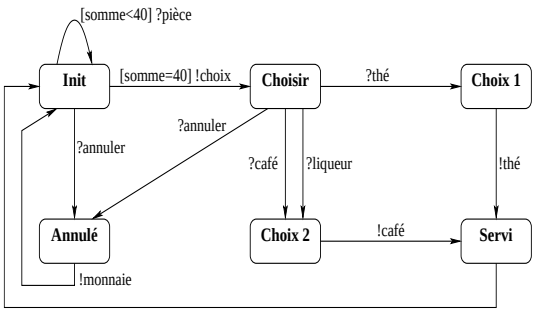


Fig. 1.4 : *Spécification d'un distributeur de boissons*

Cette différence entre entrées et sorties nécessite des modèles plus riches [71] que l'on appelle systèmes de transitions à entrées/sorties. Les modèles les plus utilisés [71] sont les IOA (*Input Output Automata*) [83, 106], les IOSM (*Input Output State Machine*) [96], les IOTS (*Input Output Transition System*) [115] et les IOLTS (*Input Output Labelled Transition System*) [91]. Ces graphes représentent les comportements valides du système à tester. Les tests sont obtenus à partir de chemins qui permettent de couvrir des critères comme la couverture des transitions ou des états [29], ou bien des objectifs de tests (par exemple, tester tous les cas qui permettent d'arriver à un certain noeud). Un chemin représente une séquence d'entrées-sorties (trace) qui permet de définir un scénario de test. Les entrées obtenues à partir des chemins du graphe permettent de vérifier le traitement des entrées valides alors que celles obtenues à partir de chemins n'appartenant pas au graphe permettent de vérifier la robustesse du système sur des entrées invalides.

Traditionnellement, les LTS sont finis mais il existe d'autres modèles comme les Extented LTS [79] et les IOSTS (*Input Output Symbolic Transition Systems*) [103] qui ne le sont pas. Ces spécifications formelles, comme les spécifications algébriques, qui utilisent des types de données sont plus expressives mais rencontrent en contre partie des problèmes d'indécidabilité (atteignabilité, faisabilité).

1.4 Une combinaison de méthodes de sélection : le test statistique structurel

Comme nous l'avons vu à la section 1.2, le test structurel assure la couverture d'éléments sélectionnés en un nombre fini (et souvent petit) de tests. Quant au test statistique, il permet de faire du test intensif automatisé mais au détriment d'une bonne couverture. En 1991, Waeselynck et Thévenod-Fosse [111] ont développé

une nouvelle forme de test : le test statistique structurel. L'idée de base consiste à faire du test aléatoire uniforme en introduisant des aspects structurels. Ainsi, on peut faire du test intensif tout en assurant une bonne couverture de tous les éléments structurels.

La méthode de test statistique structurel du LAAS est basée sur la construction d'une distribution du domaine d'entrée telle que la plus petite probabilité d'atteindre un élément d'un critère de couverture soit maximale et telle qu'on écarte aucun point du domaine d'entrée.

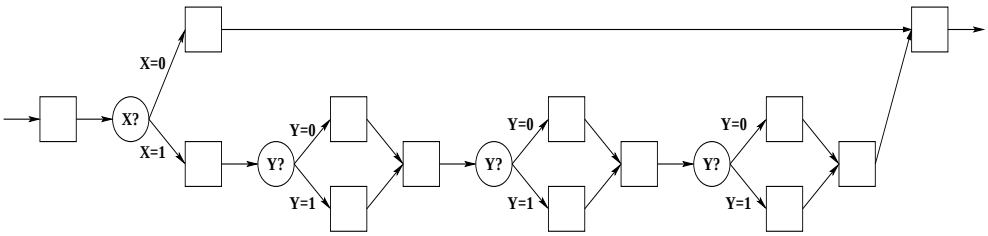


Fig. 1.5 : *Graphe de contrôle annoté du programme FCT2*

Exemple 13 (Le programme FCT2 et le critère "tous les chemins")
Nous utilisons dans cet exemple la représentation du graphe de contrôle de FCT2 de la figure 1.5. Ce programme contient 9 chemins. Les entrées du programme sont un booléen X et un bit Y. Soit $x = Prob[X = 1]$ et $y = Prob[Y = 1]$. Soit p_k, avec $k \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, la probabilité d'exécuter le chemin k. On obtient le système suivant :

$$\begin{cases} p_1 = 1 - x \\ p_2 = x.y^3 \\ p_3 = p_4 = p_5 = x.y^2.(1-y) \\ p_6 = p_7 = p_8 = x.y.(1-y)^2 \\ p_9 = x.(1-y)^3 \end{cases}$$

Une distribution optimale serait une distribution qui permettrait d'atteindre chaque chemin de manière équiprobable. On construit une telle distribution en résolvant le système suivant :

$$\begin{cases} p_1 = 1 - x = \frac{1}{9} \\ p_2 = x.y^3 = \frac{1}{9} \\ p_3 = p_4 = p_5 = x.y^2.(1-y) = \frac{1}{9} \\ p_6 = p_7 = p_8 = x.y.(1-y)^2 = \frac{1}{9} \\ p_9 = x.(1-y)^3 = \frac{1}{9} \end{cases}$$

Une distribution optimale est donc celle qui assure : $x = \frac{8}{9}$ et $y = \frac{1}{2}$.
Comme par exemple $D_X = \{0, 1, 1, 1, 1, 1, 1, 1, 1\}$ et $D_Y = \{0, 1\}$.

Cette méthode a donné de très bons résultats expérimentaux : le pouvoir de détection est bien supérieur aux méthodes purement structurelles ou purement aléatoires (distribution uniforme). Cependant toute généralisation de cette technique est freinée du fait qu'elle n'est pas totalement automatisable. En effet, la construction de la distribution nécessite la résolution d'un système d'équations qui a autant d'équations que de chemins. Par conséquent, s'il y a un nombre infini ou trop important de chemins, cette construction doit être effectuée de manière empirique, le tirage étant bien sûr automatisé. Cette construction explicite de la distribution se fait selon les cas de manière analytique (comme l'exemple 4) ou empirique :

- analytique : les conditions d'activation des éléments peuvent être exprimées en fonction des paramètres d'entrée. Dans ce cas, la construction de la distribution est basée sur la construction d'un système d'équations sur les probabilités d'activer les éléments à couvrir. Ces probabilités sont fonctions des probabilités des entrées, ce qui facilite l'obtention d'un profil qui maximise la fréquence des éléments les moins probables.
- empirique : il s'agit d'instrumenter le programme sous test dans le but de collecter statistiquement le nombre d'activations des éléments. En partant d'un grand nombre d'entrées tirées aléatoirement à partir d'une distribution initiale (en général, une distribution uniforme), on raffine progressivement jusqu'à ce que la fréquence de chaque élément soit suffisamment élevée.

Une version de cette méthode combinant test statistique et test fonctionnel a également été étudiée [117].

Le lecteur pourra trouver dans [30] un autre exemple de test statistique structurel.

Ces approches ont inspiré la méthode présentée dans cette thèse. Tout naturellement, nous retrouverons la construction d'une distribution permettant d'optimiser la plus petite probabilité d'atteindre un élément. La différence essentielle porte sur la distribution : dans notre cas, elle sera sur des chemins et non sur le domaine des entrées.

Chapitre 2

Des méthodes et des outils pour le test dynamique

Dans cette thèse, nous proposons une nouvelle approche pour le test statistique de logiciel basée sur la génération aléatoire de chemins dans un graphe. Cette approche peut s'appliquer à n'importe quelle méthode de test statistique. Selon le type de description utilisé et le critère de couverture choisi, nous obtenons une méthode de test statistique structurel ou une méthode de test statistique fonctionnel.

Bien que notre sélection soit basée sur une distribution probabiliste, notre approche est difficilement comparable aux méthodes statistiques classiques car la distribution, que nous construisons, porte sur des chemins et non sur le domaine des entrées. Nous reviendrons sur ce point au chapitre 7.

Afin de placer notre approche parmi les méthodes de test dynamique existantes, nous avons choisi de présenter un ensemble, non exhaustif, de méthodes qui utilisent la sélection de chemins et/ou des contraintes (résolution, simplification).

2.1 Sélectionner des chemins pour couvrir un critère

Si on s'intéresse aux méthodes, fonctionnelles [45, 5] ou structurelles [77, 16], basées sur des critères de couverture, les stratégies couramment adoptées dans la pratique sont basées sur la sélection d'un sous-ensemble minimal de chemins qui permet de couvrir les éléments du critère donné.

Les algorithmes consistent à choisir les chemins un à un, chaque chemin choisi passe forcément par un élément du critère qui n'est pas couvert. Le processus se répète jusqu'à la couverture totale de tous les éléments du critère.

La méthode structurelle [16] de Bertolino[1] et Marré[2] repose sur une couverture des branches d'un programme basée sur la notion de dominance[3] et sur la génération d'un ensemble de chemins qui couvrent chaque arc du graphe de flot réduit (seuls les noeuds associés à un prédicat apparaissent) du programme. Cette approche est composée de quatre étapes :

1. sélectionner un ensemble d'arcs non-contraints S. Ces arcs forment un ensemble minimum d'arcs tel que tout ensemble de chemins qui couvre cet ensemble, couvre tous les arcs du graphe de contrôle réduit.
2. choisir un arc de S non encore couvert.
3. choisir un chemin qui permet de couvrir cet arc non-contraint.
4. si tous les arcs de S sont couverts par les chemins sélectionnés s'arrêter sinon retourner à l'étape 2.

L'étape 1 peut être adaptée en fonction de la couverture recherchée et l'interaction avec un utilisateur est possible pendant l'étape 4. À l'étape 2, le choix de l'arc suivant est, par défaut, orienté par l'heuristique consistant à construire le plus petit ensemble de chemins permettant la couverture désirée. Après avoir été validée sur une étude de cas [17], cette méthode a été généralisée à n'importe quel critère de couverture [90].

Comme ces auteurs, nous nous sommes également intéressés à la notion de dominance (cf. section 5.2.2) pour déterminer une de nos premières distributions. Mais, comme nous le verrons au chapitre 4, dans notre cas, il existe une meilleure distribution.

La méthode CASTING (*Computer Assisted Software tesTING*), développée à l'IRISA Rennes par Van Aertryck [5], est très intéressante car comme notre approche, il s'agit d'une méthode applicable au test fonctionnel [5, 6] comme au test structurel. Elle s'applique aussi bien à des spécifications formelles, à du code source, à des annotations de l'utilisateur qu'à une combinaison de tous ces formalismes dès lors qu'ils sont exprimables à l'aide de grammaires attribuées [44]. La stratégie consiste à rechercher un ensemble de suites de tests les plus courtes possibles qui couvre l'ensemble des spécifications de cas de test. Une spécification de cas de test est une description d'un cas particulier d'application d'une opération : elle se compose d'une opération, d'une description du domaine d'entrée (état du système et paramètres de l'opération) et d'une description du résultat obtenu (état obtenu et sorties produites).

Pour cela, un graphe symbolique, recensant les successions de spécifications de cas de test possibles, est construit tel que :

[1]IEI-CNR, Pisa, Italie

[2]Universidad de Buenos Aires, Argentine

[3]Le lecteur trouvera page 82 une définition du terme dominance.

- les noeuds sont des représentations symboliques des états du système testé
- les arcs sont des applications des spécifications de cas de test. Il y a au moins un arc par spécification.

À partir de ce graphe, les suites de tests sont générées à l'aide de l'algorithme suivant :

1. Déterminer pour chaque spécification de cas de test la profondeur[4] de sa première application dans le graphe.
2. Choisir une spécification de cas de test non encore couverte parmi celles dont la profondeur est la plus grande.
3. Choisir le chemin le plus court qui se termine par cette spécification et qui couvre le plus possible de spécifications non couvertes.
4. Utiliser la résolution de contraintes pour instancier les variables et paramètres de chaque arc et état du chemin. Une solution constitue une suite de tests.
5. Recommencer jusqu'à ce que toutes les spécifications soient couvertes ou bien qu'une des limites soit atteinte (pourcentage des spécifications à couvrir, temps alloué, etc.).

L'outil CASTING autorise l'utilisateur à orienter, compléter ou trier les cas de test (afin de récupérer un nombre acceptable de cas de test) mais aussi à orienter la stratégie de test (choix parmi un ensemble de stratégies proposées ou définition d'une nouvelle stratégie). Une stratégie de test est composée d'une stratégie de construction du graphe symbolique et d'une stratégie de parcours de ce graphe. La construction du graphe et son parcours seront différents si l'on veut une suite de test unique qui couvre toutes les transitions, comme dans la méthode de Dick et Faivre [45], ou bien si l'on veut couvrir toutes les transitions qui amènent dans des états de blocage.

D'autres méthodes allient sélection de chemins et sélection d'entrées dans le but d'éviter ou du moins limiter la sélection de chemins infaisables. Parmi ces méthodes, on trouve les approches adaptatives[5] comme celle de Prather et Myers[6] qui est basée sur une stratégie des préfixes de chemins [98]. Le premier chemin est déterminé à partir d'un ensemble d'entrées choisies aléatoirement : il est identifié par une conjonction de prédicats $c_1 \wedge c_2 \wedge ... \wedge c_n$ qui correspondent aux noeuds de décision rencontrés le long du chemin. Le plus petit préfixe du chemin dit réversible est alors c_1, les entrées suivantes sont choisies de façon à pouvoir

[4] La profondeur est la distance entre l'état de départ de la transition qui applique une spécification de cas de test et l'état initial.

[5] Une méthode adaptative est une méthode qui adapte la sélection des tests au cours de leur exécution.

[6] Trinity University, San Antonio, USA

satisfaire $\neg c_1$. Un nouveau chemin est alors obtenu identifié par $\neg c_1 \wedge a_2 \wedge ... \wedge a_n$. Jusqu'à la couverture total des éléments du critère, on choisit à chaque étape un plus petit préfixe $b_1 \wedge b_2 \wedge ... \wedge b_k$. Ce préfixe est tel qu'en utilisant des entrées qui satisfont $b_1 \wedge b_2 \wedge ... \wedge \neg b_k$ on puisse obtenir un nouveau chemin qui couvre des éléments du critère non encore couverts.

Une autre méthode est celle proposée par Marre, Mouy et Williams du CEA Saclay [88]. Le principe consiste à choisir des entrées de test aléatoires puis d'exécuter le programme sur ces entrées. Via une instrumentation du code, le chemin symbolique, qui correspond à cette exécution, est récupéré. Le prédicat $c_1 \wedge c_2 \wedge ... \wedge c_n$ associé est obtenu à partir de ce chemin symbolique en opérant des substitutions. Ce prédicat de chemin définit le domaine d'entrée du chemin c'est-à-dire l'ensemble de toutes les entrées qui permettent de passer par le chemin exécuté.

Un "nouveau" domaine des entrées du programme est calculé par résolution de contraintes à partir du domaine de départ et du prédicat de chemin : toutes les entrées pouvant activer le chemin tiré sont écartées. Le test suivant est obtenu en choisissant des entrées dans le sous-domaine défini par le prédicat $c_1 \wedge c_2 \wedge ... \wedge \neg c_n$. L'opération est renouvelée jusqu'à la couverture de tous les chemins faisables. Le solveur de contraintes utilisé par cette méthode est une adaptation améliorée du noyau de GATeL.

2.2 Utilisations des contraintes

Les contraintes peuvent être utilisées pour caractériser les entrées qui permettent l'exécution d'un chemin [5, 90, 98, 88] mais aussi les domaines des entrées (valides ou invalides), les états d'un système ou encore l'ensemble des chemins (et indirectement des entrées) qui permettent d'atteindre un élément d'un critère, etc.

La résolution de contraintes est, quant à elle, utilisée pour calculer un élément particulier de ce qui est caractérisé. Par exemple, la résolution de contraintes caractérisant un domaine d'entrée valide permet de calculer un représentant de ce domaine.

2.2.1 Des chemins aux entrées

Comme nous l'avons déjà vu (page 26), les solveurs de contraintes diffèrent selon la nature des prédicats à résoudre. Par conséquent, ils diffèrent également selon le type de description considéré. Ainsi, lorsque CASTING est utilisé sur des spécifications B [6], le solveur utilisé est Ilog Solver, qui permet de résoudre des contraintes sur les booléens, les entiers, les rationnels et les ensembles, et lorsque

cette méthode est utilisée sur des spécifications UML [7], le solveur utilisé est écrit en SICStus Prolog en utilisant des librairies comme *clp(fd)*. Comme nous le verrons au chapitre 4, notre approche devra aussi adapter le solveur utilisé en fonction du type de description considéré.

Le solveur de contraintes que nous avons utilisé pour passer des chemins sélectionnés aux entrées dans notre prototype AuGuSTe (présenté au chapitre 6), est extrait d'un module de GATeL et utilise de la résolution randomisée comme LOFT et GATeL.

2.2.2 Caractériser le domaine des entrées

Bernot, Gaudel et Marre ([15], page 30) utilisent la résolution de contraintes de LOFT pour instancier les axiomes des spécifications algébriques.

L'outil LOFT [85, 86] (*LOgic for Functions and Testing*) a été développé en ECL^{i}PS^e [51] et utilise un programme logique construit en associant à chaque axiome une clause de Horn avec égalité. Il est basé sur la procédure de résolution déterministe, correcte et complète, de Prolog et sur un système de réécriture déterministe. Le choix du littoral à instancier se fait selon la stratégie du branchement minimal dans l'arbre de résolution (unification avec le minimum de tête de clauses) et le choix de la clause est aléatoire. Une solution est un ensemble d'instances closes des variables d'une formule équationnelle caractérisant les propriétés d'une opération.

LOFT permet à l'utilisateur :

- d'orienter la sélection en indiquant, par exemple, le nombre de dépliages souhaité pour chaque opération,
- de limiter la profondeur de l'arbre de résolution ce qui permet d'obtenir une solution en temps fini (quitte à perdre la complétude),
- de donner des stratégies qui lui permettent d'obtenir ses hypothèses de régularité et d'uniformité.

L'objectif du mécanisme de résolution, consistant à minimiser le branchement moyen de l'arbre de résolution et donc la taille des chemins menant à une solution, a été repris dans GATeL [87, 13] (Génération Automatique de séquences de Tests à partir de descriptions LUSTRE). C'est un outil d'assistance à la génération de séquences de tests à partir de descriptions en langage déclaratif de flots de données synchrones LUSTRE [66], utilisé pour le test unitaire et d'intégration, structurel ou fonctionnel. Les heuristiques utilisées par le solveur de contraintes de cet outil ont été déjà présentées au chapitre 1.

Ici, le langage LUSTRE peut être utilisé pour décrire la spécification du système à tester ou bien être le source du code qui sera ensuite compilé du programme sous test. Les systèmes considérés sont synchrones, ce qui permet de considérer des comportements cycliques et déterministes. Dans le premier cas, la

méthode de sélection de test sera plutôt fonctionnelle alors que dans le second, elle sera structurelle. La description de l'environnement du système à tester et l'objectif de test à atteindre sont également décrits en LUSTRE.

En effet, l'utilisateur peut éventuellement compléter la description du système ou de la spécification par une description de l'environnement en utilisant par exemple des assertions sur les entrées. De plus, il doit fournir un objectif de test (propriété sur les entrées/sorties du système). GATeL interprète toutes ces constructions LUSTRE par des contraintes sur des variables booléennes ou à valeur dans des intervalles d'entiers. La génération d'une séquence de tests est traitée comme la résolution de ce système de contraintes.

Les outils LOFT et GATeL ont été validés sur études de cas réalistes. LOFT a été utilisé pour dériver des jeux de test de la partie embarquée d'un système de conduite d'un métro automatique [36] et a été évalué expérimentalement par rapport à l'approche statistique de Thévenod-Fosse et Waeselynck, présentée en section 1.4, dans [89]. Une première étude de cas a été effectuée sur GATeL lors de l'action de recherche FORMA [31]. Après plusieurs études de cas internes au CEA, GATeL est maintenant utilisé pour l'évaluation des tests de logiciels critiques à l'IRSN (Institut de Radioprotection et Sûreté Nucléaire).

D'abord restreinte aux prédicats contenant des variables booléennes et/ou entières [85, 87], la résolution randomisée a été étendue aux tableaux d'entiers et/ou de booléens dans notre prototype. L'avantage de résolution randomisée est de permettre pour un prédicat donné de ne pas toujours avoir la même solution : ce qui est un atout non négligeable dans le cas du test.

Une autre méthode fonctionnelle basée sur des systèmes décrits en LUSTRE est celle développée au laboratoire LSR-IMAG de Grenoble. Cette méthode est basée sur une version exécutable du système à tester, sur une spécification partielle (non-déterministe) de son environnement, sur un oracle, sur une longueur de séquence de tests souhaitée (nombre de cycles) et un germe permettant l'initialisation du générateur aléatoire [22]. Les contraintes d'environnement décrivent des restrictions sur les valeurs possibles du vecteur d'entrées du système à tester. Puis, un diagramme de décision binaire (*Binary Decision Diagrams* ou BDD [10]) est dérivé à partir de ces contraintes : tout chemin menant à une feuille étiquetée *true* est un vecteur d'entrées valides. Une dérivation entièrement automatisée des données est effectuée à partir de l'environnement. Un générateur, qui simule l'environnement du système à tester, sélectionne un vecteur d'entrées pour ce système et lui envoie. Le système à tester réagit avec un vecteur de sorties qui est retourné au générateur. Le générateur produit alors un nouveau vecteur d'entrées en fonction de ce vecteur de sorties et le cycle se répète. L'oracle observe les entrées/sorties et détermine si les propriétés requises sont violées ou non, mais quelle que soit son observation, un test ne s'arrête que lorsque la longueur de la séquence de test souhaitée est atteinte. Cette méthode est implémentée par

l'outil LUTESS [25, 26] qui est un environnement construit autour du langage LUSTRE. La spécification partielle de l'environnement s'exprime sous forme d'expressions booléennes LUSTRE, le système à tester est un programme synchrone avec des entrées/sorties booléennes et l'oracle est un observateur synchrone décrit en LUSTRE. Dans LUTESS, à chaque cycle, la génération des vecteurs d'entrées est, par défaut, effectuée de manière aléatoire uniforme parmi tous les vecteurs d'entrées qui satisfont la spécification de l'environnement. Cependant, l'utilisateur peut choisir une autre manière de sélectionner les vecteurs d'entrées. Il peut :

- définir une distribution statistique (uniforme ou opérationnelle) des entrées [22]
- définir des propriétés de sûreté [95] et le générateur sélectionnera alors les entrées susceptibles de pousser le système à tester à violer ces propriétés
- choisir des scénarios [123] et le générateur choisit alors les entrées qui respectent ces scénarios.

L'outil LUTESS a montré son efficacité sur une étude de cas industrielle [64, 24].

LURETTE [102] est également un outil de test fonctionnel développé au Vérimag de Grenoble, qui repose sur les mêmes principes que LUTESS auxquels est rajouté un processus de contrôle du temps et donc des cycles. Cependant, contrairement à LUTESS, l'utilisateur ne peut pas guider la sélection des données. La méthode utilisée permet de tester plusieurs transitions à chaque étape : plusieurs vecteurs de données sont produites et soumises, puis en fonction des réactions du système, la transition la plus pertinente est choisie.

Il n'a pas de stratégie élaborée pour les booléens mais par contre, des techniques d'analyse statique sont appliquées pour les entiers et les réels [26]. Les noeuds du BDD sont des variables booléennes ou une abstraction des contraintes arithmétiques. En résolvant à l'aide de polyèdres les contraintes amenant à une feuille étiqueté `true`, on obtient un ensemble de solutions possibles parmi lequel on en choisit une.

Des travaux [53] sont actuellement en cours pour fusionner les outils LUTESS et LURETTE.

La méthode de Legeard, Peureux et Utting[7] [80] permet une automatisation du test fonctionnel aux limites de spécifications formelles ensemblistes en utilisant des techniques de résolution de contraintes. Les spécifications en entrée sont des spécifications Z ou B qui spécifient une seule machine avec des pré-conditions explicites (i.e. calculables) pour les opérations et dont toutes les structures de données sont finies (énumérables ou de cardinalités finies). Ces spécifications sont réécrites en des systèmes de contraintes équivalents à partir desquels un solveur de contraintes permet de construire les états limites[8] et de partitionner le domaine

[7] Laboratoire d'Informatique de l'université de Franche-Comté, Besançon

[8] État dans le lequel au moins une variable d'état a comme valeur l'un de ces extremums.

de chaque variable d'état pour en extraire des valeurs limites. Un test consiste alors à activer chaque comportement des opérations à partir d'un état limite.

Une première étape consiste à calculer l'ensemble des buts limites (*boundary goals*) à partir d'une forme normale disjonctive de la spécification. Ces buts limites sont des prédicats qui décrivent des sous-ensembles de l'espace des états. Ensuite, chaque but limite est instancié en un ou plusieurs états limites (*boundary state*) atteignables à partir de l'état initial du système. Cette instanciation est effectuée à l'aide du solveur de contraintes et permet ainsi d'obtenir des scénarios de test.

Le BZ-TT présenté dans [12] est l'outil implémentant cette méthode de test à partir d'une extension du solveur de contraintes CLPS-B [21]. Cette méthode a été validée par plusieurs études de cas industrielles [14, 28].

2.2.3 Caractériser les chemins atteignant un élément du critère

L'approche proposée [60] par Gotlieb, Botella et Rueher[9] est une approche orientée objectif, c'est-à-dire qu'ils se fixent un point (branche, instruction) relatif au critère ("tous les enchaînements", "toutes les instructions") à atteindre puis ils génèrent des entrées qui permettent d'atteindre ce point. Cette approche implémentée dans l'outil industriel INKA est composée de deux grandes étapes :

1. Le programme est transformé en un système de contraintes en utilisant le graphe de contrôle et la forme SSA (*Static Single Assignment*) [35]. La forme SSA d'une séquence de code linéaire est obtenue par simple renommage des variables comme cela se fait pour les prédicats de chemins (voir section 1.2.2). Pour les structures de contrôle, des assignations spéciales appelées ϕ-fonctions sont introduites. Une ϕ-fonction retourne l'un ou l'autre de ces arguments en fonction du flot de contrôle.

   Le système de contraintes ainsi obtenu est composé de contraintes globales du programme et de contraintes qui sont spécifiques au point à atteindre.

2. Ensuite, résoudre le système de contraintes pour savoir si au moins un chemin faisable existe (il y a une solution) et générer automatiquement les données de tests qui correspondent à l'un de ces chemins faisables.

Une méthode similaire [65] est celle développée par Gupta[10], Mathur[11] et Soffa[12] mais contrairement à la méthode précédente le système de contraintes n'est pas construit pour tout le programme mais pour un sous ensemble. Le principe

[9]Thomson-CSF, Élancourt, France et université de Nice-Sophia-Antipolis, Sophia-Antipolis, France

[10]University of Arizona, Tucson, USA

[11]Purdue university, West Lafayette, USA

[12]University of Pittsburgh, Pittsburgh, USA

de cette méthode est le suivant : Des entrées sont choisies arbitrairement parmi toutes les entrées possibles et le programme est exécuté sur ces entrées. Si le point sélectionné est atteint, c'est terminé. Sinon les entrées choisies arbitrairement sont alors raffinées à partir d'un système de contraintes calculé pour un chemin. Ce chemin est sélectionné parmi ceux qui

1. appartiennent au graphe de contrôle partiel dont seuls les sommets prédicats sont considérés
2. passent par le point sélectionné
3. n'ont pas été identifiés comme étant des chemins infaisables
4. n'ont pas été classés comme des chemins ayant une forte probabilité d'être infaisables

En cas d'échec, le processus de raffinage puis d'exécution se réitère. Le chemin sélectionné d'une itération à l'autre n'est pas forcément le même, cela dépend d'une mesure mise à jour à chaque fois.

2.2.4 Problèmes communs pour la résolution

Pour la majorité de ces méthodes de test, deux problèmes se posent pour la résolution :

– soit elle ne termine pas (indécidabilité)
– soit elle échoue

Dans le cas des domaines finis, la processus de résolution est décidable mais au prix d'une complexité ingérable (énumération des domaines). De nombreuses méthodes, notamment celles qui sont basées sur le noyau de GATeL, utilisent des systèmes de *time-out* c'est-à-dire qu'elles limitent le temps passer à essayer de résoudre un prédicat et/ou le nombre de *backtracking*.

En cas d'échec de la résolution, selon les modèles utilisés, cela correspond à des chemins dits infaisables c'est-à-dire pour lesquels il n'existe pas d'entrée permettant leur exécution ou à des états du système inatteignables. Dans ce cas, il existe plusieurs types de stratégie pour gérer ce problème.

La stratégie la plus simple, utilisée par Gotlieb, Botella et Rueher [60] et Van Aertryck [5], est de fournir l'information à l'utilisateur et de le laisser seul juge de la suite. Une autre approche, adoptée par Gupta, Mathur et Soffa [65], consiste à maintenir à jour un ensemble de chemins reconnus comme étant infaisables. Il s'agit alors de rejeter tout chemin tiré qui appartiendrait à cet ensemble. D'autres méthodes utilisent des heuristiques pour choisir des chemins qui ont plus de chance d'être faisables. Parmi ces stratégies, on trouve celle de Yates et Malevris [122] qui consiste à favoriser les chemins les moins contraints c'est-à-dire passant par le moins de points de choix possibles.

Concernant les chemins soupçonnés d'être infaisables[13], certaines méthodes, comme celle de Gupta, Mathur et Soffa [65], maintiennent un ensemble de chemins soupçonnés d'être infaisables et pour chacun de ces chemins un compteur[14] qui est incrémenté à chaque fois que la résolution du prédicat associé est indéterminée (au bout d'un certain laps de temps et d'essai). Si le compteur atteint la limite fixée, le chemin est alors considéré comme étant certainement infaisable et est déplacé dans l'ensemble des chemins infaisables.

Une autre solution est de développer des méthodes adaptatives, comme celle de Marre, Mouy et Williams [88], qui permettent de corriger la génération des tests au fur et à mesure de leur exécution et ainsi limiter ce problème.

2.3 Récapitulatif

La méthode de test statistique que nous proposons étant difficilement comparable aux méthodes classiques, nous l'avons comparée à des méthodes structurelles et fonctionnelles en fonction des deux étapes principales qui la compose :

– la sélection de chemins
– la résolution de contraintes

Parmi les méthodes basées sur la sélection de chemins, nous n'en avons trouvé aucune qui assure la couverture des chemins. La pratique courante consiste à sélectionner le plus petit sous-ensemble de chemins permettant de couvrir un critère.

Les contraintes sont utilisées pour caractériser les domaines d'entrée valides, les états d'un système ou encore les entrées qui permettront l'exécution d'un chemin. La résolution de contraintes permet d'obtenir des représentants des ensembles à caractériser. Très peu de ces méthodes échappent au problème des chemins infaisables, et plus généralement au problème indécidable d'insatisfiabilités des contraintes. Soit elles utilisent des heuristiques permettant de limiter le risque, soit elles régénèrent autant de cas de tests qu'il y a eu d'échec.

De plus, on remarque que la plupart de ces méthodes s'appliquent à un type de description particulier et rares sont celles qui peuvent facilement s'adapter.

[13]Ce sont des chemins pour lesquels la résolution est indéterminée : elle s'est arrêtée après avoir atteint les limites.

[14]Ce compteur sert à estimer la possibilité qu'un chemin a d'être infaisable.

Deuxième partie

Test de logiciel et structures combinatoires

Chapitre 3

Structures combinatoires

Comme nous l'avons vu dans le chapitre précédent, de nombreuses méthodes de test utilisent des modèles basés sur les graphes pour représenter le système à tester. Notre approche, dont le principe est la génération aléatoire uniforme de chemins exécutables, est également basée sur ces modèles.

En algorithmique, la modélisation de tels problèmes se fait à l'aide d'objets que l'on appelle des **structures combinatoires**. Une structure combinatoire est soit un objet atomique, soit un objet obtenu en combinant à l'aide d'opérateurs des objets atomiques. La taille d'un objet combinatoire est le nombre d'objets atomiques qui le composent. Tout ensemble d'objets combinatoires est dénombrable.

Un sous-ensemble de ces objets est l'ensemble des objets **décomposables** : ces objets peuvent être construits de *manière unique* à partir d'objets de taille plus petite. Les objets décomposables peuvent, en particulier, être décomposés sous forme d'une grammaire.

Il existe de nombreux travaux portant sur la construction d'objets combinatoires, leur énumération, leur analyse, leur génération (voir par exemple [32])... Dans ce mémoire, nous nous intéressons aux travaux sur la génération aléatoire.

Nous présentons dans la suite les techniques de génération aléatoire de structures combinatoires que nous utilisons puis nous terminons en expliquant comment représenter n'importe quel graphe à l'aide de spécifications combinatoires. Ce chapitre est inspiré du cours de "Combinatoire et Génération Aléatoire" du DEA Algorithmes, que j'ai suivi durant mon doctorat, ainsi que du mémoire d'habilitation [42] d'Alain Denise.

3.1 Génération de structures combinatoires

Les premiers travaux systématiques sur la génération aléatoire de structures combinatoires décomposables ont été effectués par Wilf et Nijenhuis [119, 93]. Ces travaux portaient sur la génération aléatoire *uniforme* ou *équiprobable* de

tels objets. Le problème était le suivant : *étant donné un ensemble E d'objets combinatoires et un entier n, soit E_n l'ensemble des éléments de E de taille n. Il s'agit d'engendrer aléatoirement un ou une séquence d'éléments de E_n de sorte que tous aient la même probabilité d'apparaître.*

Il existe trois grandes approches pour la génération aléatoire : la méthode récursive, les méthodes à rejet et les méthodes basées sur les chaînes de Markov ([72, 99, 100, 120]). Dans ce chapitre, nous nous intéressons aux deux premières approches.

3.1.1 La méthode récursive de génération aléatoire

3.1.1.1 Génération uniforme de structures décomposables

Les principes de base de la génération uniforme de structures décomposables ont été proposés par Wilf [119]. À l'origine, il existe une description récursive non ambiguë de l'ensemble dans lequel on veut tirer des objets combinatoires. Cette description détermine des choix à effectuer pour engendrer des objets pas à pas. Les probabilités de choix se calculent en fonction de la cardinalité des ensembles mis en jeu dans cette description récursive.

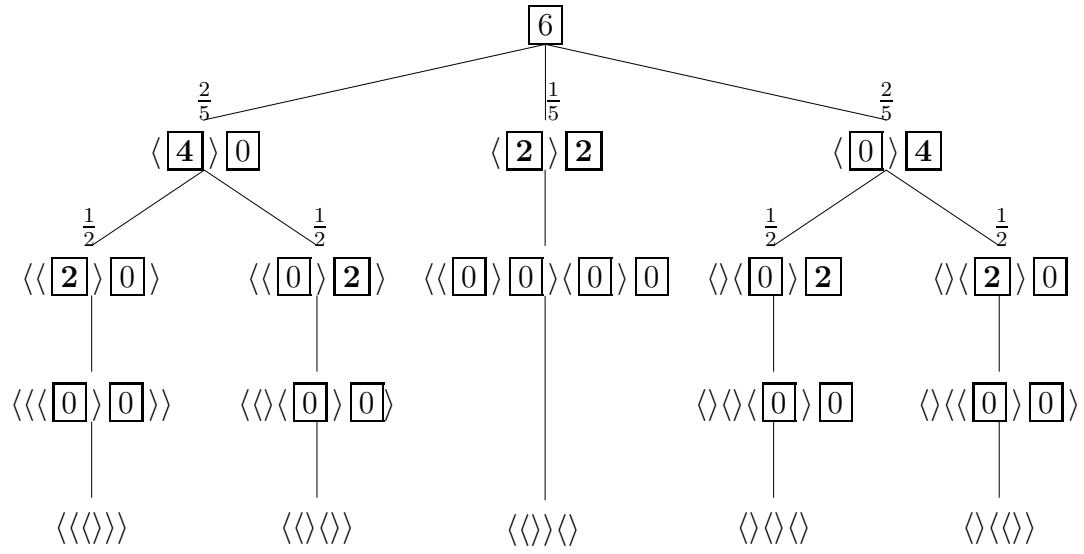


Fig. 3.1 : *Génération de mots bien parenthésés de longueur 6.*

Cette récursive est illustrée en figure 3.1. Il s'agit d'engendrer des mots bien parenthésés de longueur 6. Une description récursive non ambiguë d'un mot bien parenthésé est la suivante : soit c'est le mot vide soit il s'agit d'une parenthèse ouvrante suivie, notée $\langle$, d'un mot bien parenthésé suivi d'une parenthèse fermante,

notée $\rangle$, suivie d'un mot bien parenthésé[1]. Ce qui se définit de manière formelle par :

$$S \rightarrow \epsilon | \langle S \rangle S$$

où S est appelé *source* (*start symbol*) de la structure combinatoire. Sur la figure 3.1, les probabilités de choix sont marquées sur les arêtes et le nombre marqué dans un carré indique la longueur du mot bien parenthésé qu'il faudra engendrer.

D'abord restreinte aux objets en bijection avec des chemins particuliers d'un plan, cette approche fut ensuite étendue aux objets combinatoires qui peuvent s'exprimer de façon unique à partir d'un ensemble éventuellement infini d'objets de base et d'un opérateur de composition $\otimes$, commutatif et associatif [93]. L'algorithme proposé permet alors d'engendrer de manière uniforme des multi-ensembles d'objets combinatoires à condition de savoir engendrer uniformément les objets en question, considérés comme des objets de base.

3.1.1.2 Les spécifications combinatoires

Flajolet, Zimmermann et Van Cutsem [56] proposent ensuite un schéma général de décomposition d'objets combinatoires qui repose sur la notion de *spécification combinatoire*. Une spécification de structures combinatoires (ou spécification combinatoire) consiste en un ensemble de règles de production construites à partir d'objets de base et d'opérateurs.

Les objets de base sont :
- de taille 0 appelés *objets vides* que l'on note 1 ou ϵ
- de taille 1 appelés *atomes*, qui peuvent être étiquetés ou non

Dans ce mémoire, nous nous intéressons aux cas où **les atomes ne sont pas étiquetés**.

Les opérateurs sont :
- l'union disjointe $+$
- le produit (non commutatif) noté $\times$, où $A \times B$ représente l'ensemble des couples composés d'un élément de A et d'un élément de B.
- la séquence notée *sequence(..)*, où *sequence*(A) représente l'ensemble des suites finies d'éléments de A.
- l'ensemble noté *set(..)*, où *set*(A) représente l'ensemble des ensembles finis d'éléments de A.
- le cycle noté *cycle(..)*, où *cycle*(A) représente l'ensemble des cycles finis d'éléments de A.

[1] Le langage des mots biens parenthésés est aussi connu sous le nom de langage de Dyck

Définition 9 (Spécification combinatoire [56])
Soit $T = (T_0, T_1, ..., T_m)$ une famille de $m+1$ ensembles d'objets combinatoires étiquetés. Une spécification combinatoire de T est un ensemble de $m+1$ équations telles que la ième (pour tout $0 \leq i \leq m$) s'écrit :

$$T_i = \Psi_i(T_0, T_1, ..., T_m)$$

où Ψ_i est une combinaison des cinq opérateurs définis ci-dessus appliquée à l'objet vide, aux atomes et aux T_i.

Exemple 14 (Spécification combinatoire(1)) *Soit F un atome.*
Une spécification combinatoire correspondant aux arbres binaires complets[2] *dont les feuilles sont des atomes est :*

$$A = F + A \times A$$

ou en utilisant les notations de la définition :

$$\begin{cases} T = (T_0, T_1, T_2) \\ T_0 = A = +(T_1, T_2) \\ T_1 = F \\ T_2 = \times(T_0, T_0) \end{cases}$$

Exemple 15 (Spécification combinatoire(2)) *Soit S un atome.*
Une spécification combinatoire correspondant aux arbres binaires complets dont les sommets sont des atomes est :

$$A = S + S \times A \times A$$

L'opérateur *sequence* (resp. *set* et *cycle*) peut être accompagné d'un argument qui fixe une condition sur la cardinalité des suites (resp. ensembles, cycles).

Exemple 16 (Cardinalité sur les opérateurs) *Soit F un atome.*
Une spécification combinatoire correspondant aux arbres **n***-aires complets dont les feuilles sont des atomes est :*

$$A = F + \textit{sequence}(A, \textit{card}=\mathbf{n})$$

Une spécification combinatoire correspondant aux arbres complets dont chaque sommet interne a **au plus n** *fils et dont les feuilles sont des atomes est :*

$$A = F + \textit{sequence}(A, \textit{card}\leq\mathbf{n})$$

[2]Chaque sommet d'un arbre binaire complet est soit une feuille soit un noeud avec 2 branches.

Parmi les relations qui existent entre les opérateurs, celle qui nous intéressera le plus par la suite est celle qui relie le produit $\times$ et la séquence :

$$sequence(A, card{=}k) \equiv \underbrace{A \times A \times ... \times A}_{k fois}$$

$$sequence(A, card{\leq}k) \equiv \epsilon + A + ... + \underbrace{A \times A \times ... \times A}_{k fois}$$

Un objet de taille k d'une spécification combinatoire est une suite de k atomes issue de cette spécification.Par exemple, dans le cas de la spécification combinatoire correspondant aux arbres binaires complets dont les feuilles sont des atomes, la taille d'un objet de cette spécification est le nombre de feuilles. Ainsi si F est un atome, l'objet combinatoire $F \times F \times F \times F \times F$ est de taille 5.

3.1.1.3 Des spécifications combinatoires aux spécifications standard

Pour obtenir des algorithmes de génération aléatoire efficaces, il est nécessaire de transformer les spécifications combinatoires en spécifications standard : c'est notamment le cas pour les grammaires algébriques.

Définition 10 (Spécification standard[56])
Soit $T = (T_0, T_1, ..., T_m)$ une famille de $m+1$ ensembles d'objets combinatoires étiquetés. Une spécification standard de T est un ensemble de $m+1$ équations telles que la ième (pour tout $0 \leq i \leq m$) s'écrit :

$$T_i = 1 \text{ ou } T_i = Z \text{ ou } T_i = U_j + U_k \text{ ou } T_i = U_j \times U_k \text{ ou } \Theta T_i = U_j \times U_k$$

où chaque $U_j \in \{1, Z, T_0, ..., T_m, \Theta T_0, ..., \Theta T_m\}$ et où le symbole Θ est un opérateur de pointage[3] *défini de la manière suivante :*

$$\Theta A = \bigcup_{n=1}^{\infty} (\mathcal{A}_n \times \{1, 2, .., n\})$$

avec $\mathcal{A}_n$ l'ensemble des objets de A de taille n.

Théorème 1 (Flajolet, Zimmermann, Van Cutsem[56])
Tout ensemble qui admet une spécification combinatoire admet une spécification standard. ♢

Pour tout détail sur la transformation d'une spécification combinatoire en une spécification standard, nous renvoyons le lecteur à l'article de Flajolet, Zimmermann et Van Cutsem [56].

[3] ou d'étiquetage

3.1.1.4 Algorithme de génération

L'algorithme de génération découle de la spécification standard et se décompose en deux grandes étapes :

1. une étape de dénombrement
2. une étape de génération

Dénombrement Soit n la taille des objets à engendrer. L'étape de dénombrement consiste à calculer pour tout ensemble E intervenant dans la spécification standard (i.e. les T_j) et pour tout $0 \leq i \leq n$, le nombre e_i d'objets de taille i de E. C'est le coefficient de t^k dans la série génératrice :

$$E(t) = \sum_{k \geq 0} e_k t^k$$

Les coefficients e_i peuvent être calculé en utilisant les équations de récurrence suivantes :

$$\begin{array}{lcl}
E = \epsilon & \Rightarrow & e_0 = 1 \\
E = Z & \Rightarrow & e_1 = 1 \\
E = A + B & \Rightarrow & e_n = a_n + b_n \\
E = A \times B & \Rightarrow & e_n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) a_k b_{n-k} \\
\Theta E = A \times B & \Rightarrow & e_n = \frac{1}{n} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) a_k b_{n-k} \\
E = \Theta A & \Rightarrow & e_n = n a_n
\end{array}$$

La phase de dénombrement n'est effectuée qu'une seule fois quel que soit le nombre d'objets de taille inférieure ou égale à n que l'on veut engendrer.

Exemple 17 (Dénombrement et bon parenthésage)
Supposons que l'on veuille engendrer des expressions bien parenthésées de longueur $n = 8$.
Une spécification combinatoire correspondante est

$$S = \epsilon + \langle \times S \times \rangle \times S$$

Une spécification standard associée est :

$$\begin{array}{lcllcl}
S & = & A + B & D & = & S \times E \\
A & = & \epsilon & E & = & F \times S \\
B & = & C \times D & F & = & \rangle \\
C & = & \langle &
\end{array}$$

Les équations générales pour l'algorithme de dénombrement sont :

$$\begin{cases} s_n &= a_n + b_n \\ a_0 &= 1 \\ a_n &= 0 \quad \forall n \neq 0 \\ c_1 &= 1 \\ c_n &= 0 \quad \forall n \neq 1 \\ f_1 &= 1 \\ f_n &= 0 \quad \forall n \neq 1 \end{cases} \qquad \begin{cases} b_n &= \sum_{k=0}^{n} c_k d_{n-k} \\ d_n &= \sum_{k=0}^{n} s_k e_{n-k} \\ e_n &= \sum_{k=0}^{n} f_k s_{n-k} \end{cases}$$

Dans le cas de la génération d'objets de taille 8, on a alors le système suivant :

	n=0	*n=1*	*n=2*	*n3*	*n=4*	*n=5*	*n=6*	*n=7*	*n=8*
s_n	*1*	*0*	*1*	*0*	*2*	*0*	*5*	*0*	*14*
a_n	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
b_n	*0*	*0*	*1*	*0*	*2*	*0*	*5*	*0*	*14*
c_n	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
d_n	*0*	*1*	*0*	*2*	*0*	*5*	*0*	*14*	*0*
e_n	*0*	*1*	*0*	*1*	*0*	*2*	*0*	*5*	*0*
f_n	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Génération L'étape de génération s'effectue de manière récursive : Soit E un ensemble d'objets et soit n la taille de l'objet à engendrer.

Si $E = 1$ **et** $n = 0$ alors renvoyer 1 sinon échec

Si $E = Z$ **et** $n = 1$ alors renvoyer Z sinon échec

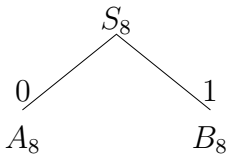
Si $E = A + B$ alors
- tirer uniformément un réel r entre 0 et 1
- si $r \leq \frac{a_n}{e_n}$ alors engendrer un objet de A taille n sinon engendrer un objet de B de taille n

Si $E = A \times B$ alors
- tirer uniformément un réel r entre 0 et 1
- choisir la plus petite taille $k \leq n$ telle que la probabilité d'engendrer un objet de taille inférieure ou égale à k à partir de A soit supérieure à r i.e. $r \leq \sum_{j=0}^{k} \frac{a_j b_{n-j}}{e_n}$
- engendrer un objet de A de taille k
- engendrer un objet de B de taille $n - k$
- concaténer les deux objets

Si $E = \Theta A$ (resp. $\Theta E = A \times B$) alors engendrer un élément de A (resp. $A \times B$) de taille n puis lui associer un nombre entre 1 et n (resp. effacer le nombre associé)

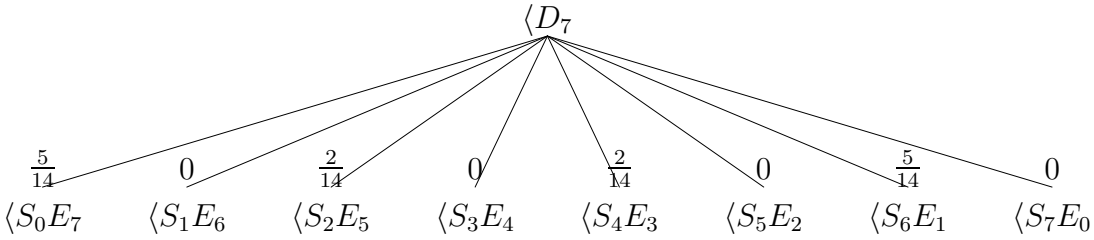
Exemple 18 (Génération et bon parenthésage)
On désire engendrer des objets de taille $n = 8$.

À l'étape 1, on a $S = A + B$ avec une probabilité nulle d'engendrer des objets de longueur 8 avec A car $\frac{a_8}{s_8} = \frac{0}{14}$. On choisit donc d'engendrer des objets de taille 8 à partir de B.



À l'étape 2, on a $B = C \times D$ et $n = 8$. Le seul k possible est 1 car à partir de C, on ne peut engendrer exactement qu'un seul objet de taille 1. Il reste donc à engendrer un objet de taille $n - k$ soit 7 à partir de D.

À l'étape 3, on a $D = S \times E$ et $n = 7$. Supposons que l'on ait tiré $r = \frac{2}{5}$, alors la taille k qui convient est $k = 2$. Il s'agit maintenant d'engendrer un objet de taille 2 à partir de S, ce qui se fait facilement car il n'y a qu'une seule possibilité, et un objet de taille 5 à partir de E.



À l'étape 4, on a $E = F \times S$ et $n = 5$. Le seul k possible est 1 car à partir de F, on ne peut engendrer exactement qu'un seul objet de taille 1. Il reste donc à engendrer un objet de taille 4 à partir de S.

À l'étape 5, on a $S = A + B$ et $n = 4$. La probabilité d'engendrer un objet de taille 4 à partir de A étant nulle, cela revient à engendrer un objet de taille 4 à partir de B.

À l'étape 6, on a $B = C \times D$ et $n = 4$. Le seul k possible est 1 car à partir de C, on ne peut engendrer exactement qu'un seul objet de taille 1. Il reste donc à engendrer un objet de taille 3 à partir de D.

À l'étape 7, on a $D = S \times E$ et $n = 3$. Supposons que l'on ait tiré $r = \frac{1}{3}$, alors la taille k qui convient est 0. Il s'agit maintenant d'engendrer un objet de taille 0 à partir de S, ce qui se fait facilement car il n'y a qu'une seule possibilité c'est ϵ, et un objet de taille 3 à partir de E.

À l'étape 8, on a $E = F \times S$ et $n = 3$. Le seul k possible est 1 car à partir de F, on ne peut engendrer exactement qu'un seul objet de taille 1. Il reste donc à

engendrer un objet de taille 2 à partir de S. L'algorithme se termine alors car il n'y a qu'une seule construction possible d'un objet de taille 2 à partir de S.

La génération aléatoire d'objets de longueur n s'étend facilement aux objets de longueur $\leq n$. Pour cela, il suffit de choisir un atome a qui n'appartient pas aux atomes de la spécification et de modifier la source S de la façon suivante :

$$S = A \rightarrow \begin{cases} S' = a \times S' + S \\ S = A \end{cases}$$

où A est une expression et S' la nouvelle source de la spécification. Tout objet de longueur n engendré se compose d'un certain nombre $k \geq 0$ d'atomes a, ce qui, si on ignore les occurrences de cet atome permet d'obtenir (toujours uniformément) un objet de longueur $n - k$ donc un objet de longueur $\leq n$.

Exemple 19 (Bon parenthésage de longueur inférieure ou égale à 4)
La transformation de la spécification combinatoire donne :

$$\begin{cases} S' = a \times S' + S \\ S = \epsilon + \langle \times S \times \rangle \times S \end{cases}$$

À partir de cette spécification, on peut engendrer uniformément des objets de taille 4 comme $a \times a \times \langle \times \epsilon \times \rangle \times \epsilon$ qui correspond à l'objet $\langle\rangle$ de taille 2.

Schéma général et complexité En résumé, le schéma général de cette approche est le suivant :

1. Compter tous les objets de taille 1 à n dérivant de chaque non terminal
2. Génération : choisir à chaque étape une dérivation avec les probabilités calculées/données

L'étape de dénombrement est de complexité $\Theta(n^2)$ (en nombre d'opérations arithmétiques effectuées) mais s'il existe une formule de récurrence linéaire à coefficients polynomiaux qui donne les termes de la série génératrice (et que l'on sait calculer) alors les termes jusqu'à l'ordre n peuvent être calculés en $O(n)$ opérations arithmétiques. C'est le cas par exemple lorsque l'on tire des mots de langages algébriques.

L'étape de génération est quant à elle dans le pire des cas et en moyenne en $\Theta(n^2)$ à cause du cas $E = A \times B$. En effet, dans ce cas, choisir[4] les tailles des objets A et B peut nécessiter dans le pire des cas $\Theta(n^2)$ opérations arithmétiques.

Si l'on veut engendrer un grand nombre d'objets de grande taille, il faut s'intéresser à la complexité de l'algorithme utilisé. Un algorithme qui génère un

[4]Choix conditionné par la taille k, voir l'algorithme de génération (page 53)

objet de taille n avec une complexité en $\Theta(n^2)$, comme c'est le cas pour l'algorithme général que l'on vient de présenter, peut être trop coûteux. Dans leur article [56], Flajolet, Zimmermann et Van Cutsem décrivent un moyen qui permet d'améliorer considérablement la complexité en temps.

Théorème 2 ([56]) *La génération aléatoire d'une structure décomposable de taille n s'effectue en $O(n \log n)$ opérations arithmétiques dans le pire des cas, après un traitement préliminaire en $O(n^2)$ opérations arithmétiques. La complexité arithmétique en espace est en $O(n)$.* ♢

Dans les cas où les nombres mis en jeu sont bornés, la complexité arithmétique[5] est réaliste mais dès que ce n'est plus le cas, elle ne permet plus de refléter le temps et la mémoire utilisés par un algorithme. Lorsque les nombres sont grands, la complexité binaire (ou logarithmique) est meilleure. Cette complexité repose sur les hypothèses suivantes : pour un nombre n, la place occupée est de l'ordre de $\log n$ et une opération arithmétique simple s'effectue en temps $O(M(n))$ où $M(n)$ résulte de l'algorithme utilisé pour cette opération.

Denise et Zimmermann [54] améliorent la génération aléatoire et uniforme de structures combinatoires décomposables, étiquetées ou non. La complexité binaire en temps et en espace est alors en moyenne en $O(n^{1+\epsilon})$ après une phase de calcul de complexité binaire en temps de $O(n^{2+\epsilon})$.

Une implémentation de cette méthode de génération a été réalisée dans le package CS [43, 33], du logiciel MuPAD [109]. Nous renvoyons le lecteur au chapitre 6 pour plus de détails sur cet outil.

3.1.1.5 Le cas particulier des langages algébriques ou rationnels

Les langages algébriques, lorsqu'ils ne sont pas ambigus, constituent des structures décomposables très particulières. Leur spécification est faite par des grammaires qui n'utilisent que deux opérateurs : l'union disjointe et la concaténation. Les symboles non-terminaux de ces grammaires peuvent être assimilés à des ensembles d'objets d'une spécification combinatoire.

Dans leur article [69], Hickey et Cohen se sont intéressés à la génération aléatoire de mots issus de grammaires algébriques non ambiguës et non normalisées. Ils ont proposé deux versions de leur algorithme :

- la première effectuait un dénombrement en $O(n^{m+1})$ (en temps et en espace) où m désigne le nombre de non terminaux de la grammaire, et une génération en $O(n)$
- la seconde dénombrait en $O(n^2 \log n)$ et générait en $O((n \log n)^2)$ des structures combinatoires.

[5] Cette complexité est basée sur l'hypothèse que la place occupée par un nombre est constante et qu'une opération arithmétique simple s'effectue en temps constant.

Pour standardiser une grammaire algébrique, il suffit de la mettre sous forme normale de Chomsky c'est-à-dire telle que tout membre droit d'une règle est soit le mot vide (i.e. ϵ), soit un symbole terminal, soit un symbole non terminal, soit la concaténation de deux symboles non terminaux. Une grammaire algébrique est définie par $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R})$ où $\mathcal{V}$ est un ensemble de symboles terminaux, $\mathcal{N}$ un ensemble de symboles non-terminaux, S un symbole non-terminal particulier appelé source (ou un axiome) et $\mathcal{E}$ un ensemble de règles définies sur $\mathcal{N} \times (\mathcal{V} \cup \mathcal{N})^*$. Pour transformer une grammaire algébrique G en forme normale de Chomsky, il suffit d'itérer les règles suivantes jusqu'à obtenir un point fixe :

– Pour toutes les règles de la forme

$$A \rightarrow a_1 a_2 a_3 ... a_n \text{ avec } \forall i, a_i \in \mathcal{V}$$

substituer tous les terminaux a_i par les non terminaux A_i correspondant et rajouter n règles de symboles non-terminal A_n pour obtenir l'ensemble suivant de règles :

$$\begin{cases} A \rightarrow A_1 A_2 A_3 ... A_n \\ A_1 \rightarrow a_1 \\ A_2 \rightarrow a_2 \\ ... \\ A_n \rightarrow a_n \end{cases}$$

– Pour toutes les règles de la forme

$$X \rightarrow X_1 X_2 X_3 ... X_n \text{ avec } n \geq 2 \text{ et } \forall i, X_i \in \mathcal{N}$$

Si $n = 2$, conserver cette règle sinon introduire $n-2$ symboles non-terminaux Y_i puis remplacer cette règle par l'ensemble suivant de $n-1$ règles :

$$\begin{cases} X \rightarrow X_1 Y_1 \\ Y_1 \rightarrow X_2 Y_2 \\ Y_2 \rightarrow X_3 Y_3 \\ ... \\ Y_{n-2} \rightarrow X_{n-1} X_n \end{cases}$$

Dans le cas particulier des langages algébriques, l'algorithme général (présenté en section 3.1.1.4) offre des complexités bien plus intéressantes puisqu'il permet de dénombrer en $O(n)$ et de générer en $O(n \log n)$ des structures combinatoires.

Quant aux langages rationnels, ils n'ont pas besoin d'être mis sous la forme de Chomsky. En effet, on peut effectuer la génération par construction d'un chemin dans un automate du langage. De plus, ils se comportent de façon idéale :

Théorème 3 (Hickey et Cohen[69]) *La génération d'un mot de longueur n d'un langage rationnel s'effectue en complexité arithmétique $O(n)$ pour la phase de dénombrement, la phase de génération et la mémoire occupée.* ♢

D'après les travaux de Goldwurm[59], l'espace mémoire peut même être en $O(1)$.

3.1.2 Méthodes de génération aléatoire à rejet

Les méthodes à rejet sont en général utilisées dans les cas où on ne sait pas engendrer directement un objet combinatoire d'un ensemble donné. Le principe général est le suivant : *Soit E un ensemble d'objets combinatoires. Soit F un ensemble qui inclut E et dans lequel il est possible de faire de la génération aléatoire. Alors pour engendrer un objet aléatoire dans E, il suffit d'engendrer une suite d'éléments de F jusqu'à atteindre un élément de E.*
Tout tirage uniforme dans F est uniforme dans E. Le nombre moyen d'essais pour atteindre un élément de E est de $\frac{|F|}{|E|}$.

De plus, même pour des structures décomposables, ce type de méthode peut parfois être une alternative plus efficace que la méthode récursive.

Il existe d'autres méthodes de génération basées sur le rejet comme :

- l'approche du rejet anticipé ou "méthode florentine" qui consiste à effectuer le rejet non pas une fois que l'objet combinatoire est construit mais en cours de construction
- l'approche d'extraction-rejet de Schaeffer [104, 105]
- l'approche basée sur le modèle de Boltzmann[6] [46] où tout objet décomposable de l'ensemble, quel que soit sa taille, a une probabilité non nulle d'être engendré. Un paramètre permet de faire varier la loi de probabilité qui fixe la taille moyenne des structures engendrées.

Les méthodes basées sur l'extraction/rejet et le modèle de Boltzmann peuvent s'avérer très efficaces surtout lors de la génération d'objets dont la taille n'est pas fixe mais appartient à un intervalle donné. Elles peuvent également être utilisées dans d'autres cas que la génération aléatoire à rejet : en particulier, le lecteur trouvera dans la partie perspectives de ce mémoire un paragraphe consacré à une utilisation de la méthode de Boltzmann pour nos travaux. Cependant, deux objets de tailles différentes ont des probabilités différentes d'être engendrés et deux objets de même taille ont la même probabilité.

3.2 Le graphe orienté comme structure combinatoire

Dans cette section, nous nous intéressons à une famille particulière de graphes orientés. Un graphe est défini par un ensemble de sommets, un ensemble d'arcs et un ensemble particulier de sommets qui contient un sommet de départ[7] *Init* et

[6] Les probabilités associées aux objets engendrés s'écrivant de manière similaire à celles des configurations d'énergie en physique statistique qui correspondent à des travaux de Boltzmann.

[7] S'il y en a plusieurs, on peut facilement se ramener à un seul sommet de départ à l'aide de transitions particulières.

un sommet de sortie[8] *Exit*. On considère que chaque sommet du graphe est atteignable et qu'à partir de n'importe quel sommet du graphe, on peut atteindre un sommet de sortie. Le graphe de la figure 3.2 est un exemple de graphes considérés. Les LTS et graphes de contrôle[9] présentés au chapitre 1 font parti de cette fa-

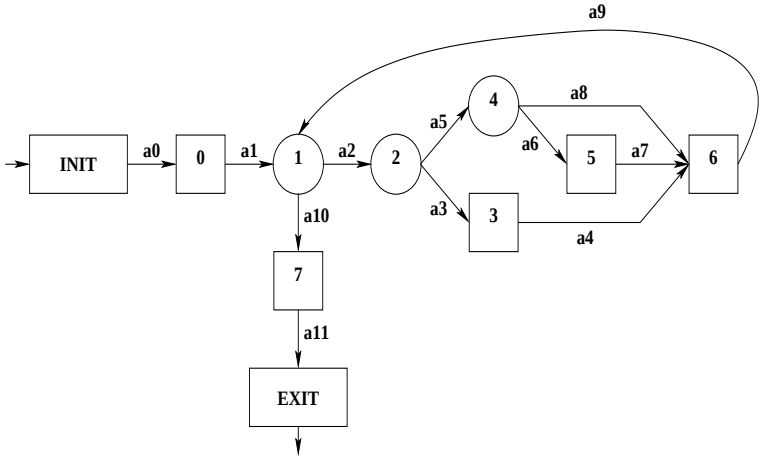


Fig. 3.2 : *Graphe de contrôle d'un programme (cf. exemple 28 du chapitre 6)*

mille. On peut les voir comme des automates finis reconnaissant les mots d'un langage rationnel. Un mot est alors soit un comportement valide, soit un chemin d'exécution possible.

Ces graphes sont des objets particuliers qui peuvent être définis de manière récursive à partir du sommet de départ. Intuitivement, on peut voir chaque T_i comme l'expression de l'ensemble des chemins qui vont du sommet Si au sommet de sortie $Exit$. Dans un premier temps, nous allons nous intéresser aux arcs comme objets atomiques, ensuite nous prendrons les sommets.

Fig. 3.3 : *Sommet seul*

[8]S'il y en a plusieurs, on peut facilement se ramener à un seul sommet de sortie à l'aide de transitions particulières.

[9]En prenant comme hypothèse qu'il n'y a pas de code mort.

Les arcs comme objets atomiques Si Si est le sommet $Exit$ (cf. figure 3.3) alors l'équation associée est :

$$T_i = \epsilon$$

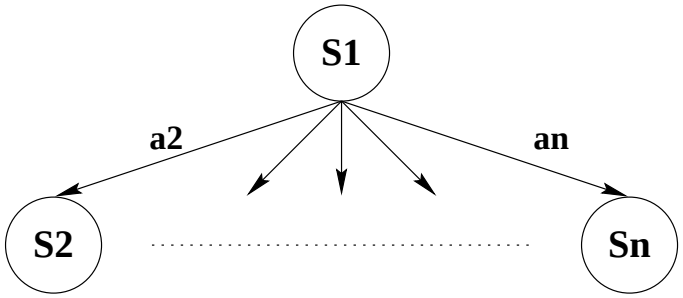


Fig. 3.4 : *Sommet relié à plusieurs sommets*

Dans les autres cas (Figure 3.4), on peut à partir du sommet $S1$, arriver à plusieurs sommets $S2$, ..., Sn en passant respectivement par $a2$,..., an. L'ensemble d'équations associé est :

$$\begin{cases} T_1 = +(\times(a2, T_2), ..., \times(an, T_n)) \\ T_2 = ... \\ ... \\ T_n = ... \end{cases}$$

Ainsi, la spécification combinatoire, dont les objets atomiques sont les arcs, qui correspond au graphe de la figure 3.2 est :

$$\begin{cases} T = \times(a0, T_0) \\ T_0 = \times(a1, T_1) \\ T_1 = +(\times(a2, T_2), \times(a10, T_7)) \\ T_2 = +(\times(a3, T_3), \times(a5, T_4)) \\ T_3 = \times(a4, T_6) \\ T_4 = +(\times(a8, T_6), \times(a6, T_5)) \\ T_5 = \times(a7, T_6) \\ T_6 = \times(a9, T_1) \\ T_7 = \times(a11, T_{EXIT}) \\ T_{EXIT} = \epsilon \end{cases}$$

Les sommets comme objets atomiques Si Si est le sommet $Exit$ (cf. figure 3.3) alors l'équation associée est :

$$T_i = Exit$$

Dans les autres cas (Figure 3.4), on peut à partir du sommet $S1$, atteindre plusieurs sommets $S2$, ..., Sn. L'ensemble d'équations associé est :

$$\begin{cases} T_1 = \times(S1, +(T_2, ..., T_n)) \\ T_2 = ... \\ ... \\ T_n = ... \end{cases}$$

La spécification combinatoire, dont les objets atomiques sont les sommets, qui correspond au graphe de la figure 3.2 est :

$$\begin{cases} T = \times(Init, T_0) \\ T_0 = \times(0, T_1) \\ T_1 = \times(1, +(T_2, T_7)) \\ T_2 = \times(2, +(T_3, T_4)) \\ T_3 = \times(3, T_6) \\ T_4 = \times(4, +(T_5, T_6)) \\ T_5 = \times(5, T_9) \\ T_6 = \times(6, T_1) \\ T_7 = \times(7, T_{EXIT}) \\ T_{EXIT} = EXIT \end{cases}$$

Au chapitre 2, nous avons vu que les graphes orientés pouvaient être utilisés pour modéliser des systèmes à tester et que des méthodes de tests cherchaient à couvrir certains éléments de ce graphe. Nous proposons de représenter l'ensemble des chemins (resp. traces) de ces modèles sous forme de structures combinatoires et de profiter des méthodes efficaces présentées dans ce chapitre pour générer aléatoirement et uniformément des chemins (resp. traces). Le chapitre 4 présente la première approche, à notre connaissance, qui combine structures combinatoires et méthodes de test.

Chapitre 4

Test statistique et structures combinatoires

Comme nous l'avons vu dans les chapitres précédents, les graphes sont souvent utilisés pour décrire le ou les comportements du système à tester. Cette description peut se faire à deux niveaux :

- au niveau de la spécification : LTS.
- au niveau de l'implémentation : graphe de flot de contrôle, de flot de données, graphe d'appels entre sous-programmes, etc.

Notre approche est basée sur un tirage de chemins parmi un ensemble de chemins qui permet de couvrir un critère donné (comme par exemple, tous les sommets). La sélection des chemins est faite en utilisant les structures combinatoires et les outils de génération aléatoire présentés au chapitre précédent.

Ce chapitre est une présentation générale de notre approche. Nous y aborderons les points délicats que sont la relation entre le nombre de test N et la qualité de test, la longueur des chemins considérés, les types de critère et l'obtention d'une distribution. Il est suivi d'un chapitre présentant le cas particulier du test structurel statistique.

4.1 Paramètres de l'approche

4.1.1 Qualité de test et nombre de tests

On a vu dans le chapitre 1 que dans le cas des méthodes où la sélection des données d'entrée est obtenue de manière déterministe, un critère de couverture C est satisfait par un jeu de test si chaque élément de C est exécuté par au moins un test. Dans le cas des méthodes statistiques, la qualité d'une méthode de test se mesure par la probabilité minimale de couvrir un des éléments du critère de test considéré. La probabilité minimale qu'a un élément d'être atteint lors d'une

série de N tests est appelée *qualité de test*. Cette notion de qualité de test pour les méthodes de test statistique a été définie par Pascale Thévenod-Fosse [110].

Nous nous inspirons de cette définition mais en l'adaptant à notre contexte c'est-à-dire en rajoutant comme paramètre la description considérée.

Définition 11 (Qualité de test)
Soit D une description d'un système à tester et C un critère de couverture.
Soit $E_C(D)$ l'ensemble des éléments caractérisé par le critère de couverture C pour la description D.
Un critère C est couvert par une méthode de test pour une description D si chaque élément de $E_C(D)$ a une probabilité d'au moins $q_{C,N}(D)$ d'être exécuté durant une exécution de N tests. La probabilité $q_{C,N}(D)$ est appelée **qualité de test**.

Ici, D peut être une spécification ou un programme, cela dépend du type de test (fonctionnel ou structurel) auquel on s'intéresse, mais D est forcément basée sur un graphe (ou plus généralement une structure combinatoire). À partir de ce graphe, il est possible de définir des critères de couverture par exemple, "tous les sommets", "tous les arcs", "tous les chemins" correspondant à une certaine propriété ("de longueur inférieure ou égale à n", "de longueur entre n_1 et n_2" ou encore "qui passent au plus m fois par chaque circuit du graphe"), etc.

Le calcul de $q_{C,N}(D)$ se fait facilement dès que $q_{C,1}(D)$ est connue. En effet, la probabilité $q_{C,N}(D)$ d'atteindre au moins une fois un élément est égale à 1 moins la probabilité de ne pas l'atteindre N fois. Ce qui se formule de la manière suivante :

$$q_{C,N}(D) = 1 - (1 - q_{C,1}(D))^N \tag{4.1}$$

La relation entre $q_{C,N}(D)$ et N peut être utilisée dans deux sens :

- Étant donné un nombre de tests, on peut calculer la qualité de test qui sera obtenue avec un tel jeu.
- Étant donné une qualité de test, on peut calculer le nombre minimum de tests à effectuer pour l'obtenir. Pour cela, il suffit d'utiliser l'inéquation :

$$N \geq \frac{\log(1 - q_{C,N}(D))}{\log(1 - q_{C,1}(D))}$$

Par la suite, lorsque le critère C et la description D seront évidents, nous utiliserons la notation q_N pour $q_{C,N}(D)$.

Prenons l'exemple de la figure 4.1, et considérons le critère de couverture "tous les chemins". On notera $\mathcal{P}$ l'ensemble de tous les chemins. Si l'on suppose les chemins équiprobables, la qualité de test est alors la suivante :

$$q_N = 1 - (1 - \tfrac{1}{|\mathcal{P}|})^N$$

ou encore puisqu'il y a 5 chemins

$$q_N = 1 - (1 - \tfrac{1}{5})^N$$

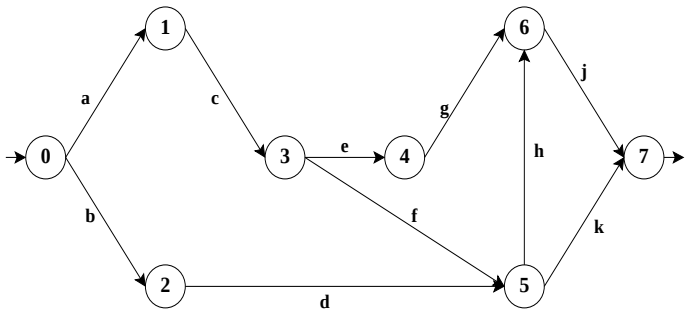


Fig. 4.1 : *Un graphe quelconque*

Avec un test, nous obtenons une qualité de 0.2 alors qu'avec une série de 5 tests, cette qualité est de 0.67. Pour obtenir une qualité de test de 0.9, il faut faire au moins 11 tests.

Cet exemple est simple car il ne contient pas de circuit et par conséquent, le nombre de chemins est fini. Regardons maintenant ce qu'il en est lorsque le graphe contient un circuit.

4.1.2 Limiter la longueur des chemins

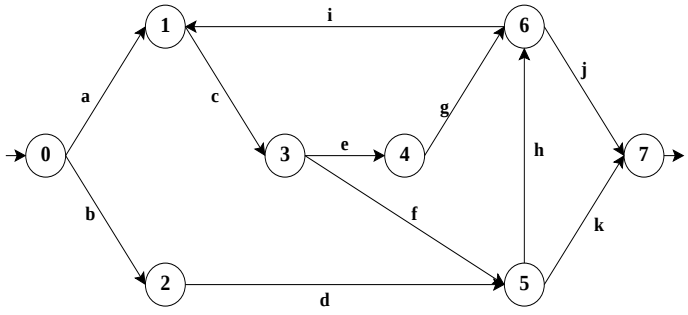


Fig. 4.2 : *Une description graphique quelconque avec boucle*

Le schéma 4.2 est celui de la figure 4.1 auquel nous avons rajouté une transition du sommet 6 au sommet 1. Ce graphe contient maintenant un circuit et l'ensemble des chemins du graphe est infini.

Deux problèmes se posent. D'abord, la présence d'une infinité de chemins rend la qualité de test q_N nulle. En effet, s'il y a une infinité de chemins, alors dans le cas d'une distribution uniforme par exemple, la probabilité d'un chemin vaut 0. Nous allons donc devoir limiter le nombre de chemins considérés[1]. Parmi les

[1]Comme dans la majorité des méthodes de test structurelles

différentes possibilités, une solution est de limiter la longueur des chemins en se donnant une longueur maximum de chemin n. Pour cela, nous rappelons que nous ne voulons que des chemins complets[2] et si possible tous les chemins élémentaires (c'est-à-dire sans circuit). Par conséquent, n doit être plus grand que la longueur du plus grand chemin élémentaire du graphe. Dans le cas de la figure 4.2, la longueur n doit être supérieure ou égale à 5.

On remarquera que le nombre de chemins croît rapidement avec n. C'est pourquoi, le choix de n doit être un compromis entre :

- choisir cette longueur suffisamment grande pour pouvoir couvrir au moins les chemins élémentaires ;
- la choisir suffisamment petite pour éviter d'avoir trop de chemins à considérer et rendre impraticable le test.

De base, on choisira n comme étant la longueur du plus grand chemin élémentaire. Cependant, il est possible que l'on veuille tester des chemins plus longs (par exemple, ceux qui passent plusieurs fois par une boucle). De plus, le choix de n peut avoir une influence sur la couverture du domaine d'entrée. Le paramètre n devra être choisi en considérant ces cas.

Reprenons l'exemple de la figure 4.2 en fixant n à 9, ce qui laisse une chance de passer au moins une fois par tous les circuits. Notons $\mathcal{P}_{\leq n}$ l'ensemble des chemins de longueur inférieure ou égale à n. Considérons maintenant le critère de couverture "tous les chemins de longueur ≤ 9". Nous avons alors la qualité de test suivante :

$$q_N = 1 - (1 - \frac{1}{|\mathcal{P}_{\leq 9}|})^N$$

Comme il y a 14 chemins de longueur inférieure ou égale à 9 (voir table 4.2), nous avons :

$$q_N = 1 - (1 - \frac{1}{14})^N$$

Dans ce cas, pour obtenir une qualité de test de 0.9999, on a besoin d'exécuter une série de 125 tests. Le tableau 4.1 décrit la relation entre le nombre de tests à exécuter pour obtenir les qualités de test 0.9, 0.99, 0.999 et 0.9999 ou encore les qualités de test obtenues lors de l'exécution de 32, 63, 94 et 125 tests.

q_N	0.9999	0.999	0.99	0.9
N	125	94	63	32

Tab. 4.1 : *Relation entre q_N et N pour le critère "tous les chemins de longueur inférieure ou égale à 9*

[2]Chemins complets et chemins élémentaires : voir définition 1, section 1.2.1

longueur	chemin
0	
1	
2	
3	bdk
4	acfk, bdhj
5	*acegj*, acfhj
7	**bdhicfk**
8	*acegicfk*, acfhicfk, ***bdhicegj***, **bdhicfhj**
9	*acegicegj*, *acegicfhj*, *acfhicegj*, acfhicfhj

Tab. 4.2 : *Chemins complets de longueur inférieure ou égale à 9 du graphe de la figure 4.2.*

4.1.3 Critères de couverture

Considérons maintenant les critères de couverture que sont "tous les sommets" et "tous les arcs", et une méthode de test statistique basée sur une distribution sur les chemins. Les éléments à couvrir ont généralement des probabilités différentes d'être atteints lors de l'exécution d'un test. Certains sont couverts par tous les tests, par exemple le sommet initial v_s et le sommet terminal v_e. D'autres ont une très faible probabilité. Par exemple, dans le cas de la figure 4.2, les arcs b et d apparaissent dans seulement 5 chemins de longueur inférieure ou égale à 9 alors que les arcs a et c apparaissent dans 9 de ces chemins. Ainsi, si l'on considère une méthode de test basée sur une distribution uniforme sur les chemins, alors un tirage uniforme des chemins de $\mathcal{P}_{\leq 9}$ permet d'atteindre l'arc b avec une probabilité de $\frac{5}{14}$, et l'arc a avec une probabilité de $\frac{9}{14}$.

Dans ces cas, la qualité de test $q_{C,N}$ se détermine à partir de la probabilité la plus faible. Soit $E_C(D) = \{e_1, e_2, ..., e_m\}$ l'ensemble des éléments (sommets, arcs) d'un critère. Pour chaque entier i de $[1..m]$, soit p_i la probabilité pour l'élément e_i d'être atteint durant l'exécution d'un test issu de la méthode de test statistique considérée. On obtient alors $q_{C,N}$ de la façon suivante :

$$q_{C,N}(D) = 1 - (1 - p_{min})^N \text{ où } p_{min} = min\{p_i | i \in [1..m]\} \tag{4.2}$$

Ou encore, si c'est N qui nous intéresse :

$$N \geq \frac{\log(1 - q_{C,N})}{\log(1 - p_{min})}$$

Le p_{min} correspond ici au $q_{C,1}$ de la formule 4.1. À partir de la formule 4.2, on en déduit immédiatement que pour une description donnée D et un critère

de couverture C considéré, optimiser la qualité du test statistique étant donné un nombre de test N revient à maximiser $q_{C,1}(D)$ c'est-à-dire p_{min}. Ou encore, minimiser le nombre de test N nécessaire pour obtenir une qualité de test donné, revient à maximiser $q_{C,1}(D)$ c'est-à-dire p_{min}.

4.2 Génération de chemins

4.2.1 Prise en compte des critères de couverture

Maintenant, étant donné un critère de couverture C, notre objectif est d'optimiser la qualité de test. Nous considérons deux cas selon la nature des éléments de $E_C(D)$, plus exactement selon qu'il s'agisse d'un ensemble de chemins ou non.

Si $E_C(D)$ dénote un ensemble fini de chemins du graphe, comme dans le cas du critère "tous les chemins de longueur inférieure ou égale à n", la qualité de test est optimisée si les chemins de $E_C(D)$ sont tirés uniformément, c'est-à-dire que chaque chemin a la même probabilité $\frac{1}{|E_C(D)|}$ d'être tiré. En effet, si la probabilité d'un chemin était plus grande que $\frac{1}{|E_C(D)|}$, cela impliquerait qu'il existe au moins un chemin dont la probabilité est inférieure à $\frac{1}{|E_C(D)|}$, diminuant ainsi p_{min} et $q_{C,N}$.

Considérons maintenant le cas où $E_C(D)$ n'est pas un ensemble de chemins mais un ensemble d'éléments constitutifs du graphe comme par exemple des sommets, des arcs ou des circuits. Nous devons alors maximiser la probabilité minimale d'atteindre un élément lorsqu'un chemin est tiré, ce que ne garantit pas la génération uniforme de chemins comme on l'a vu pour l'exemple de la figure 4.2.

Le tirage des chemins devient plus compliqué et nous proposons de décomposer ce tirage en deux étapes :

1. choisir aléatoirement un élément e de $E_C(D)$, selon une distribution adéquate des probabilités (cf. section 4.2.2).
2. tirer uniformément un chemin parmi tous ceux de longueur inférieure ou égale à n passant par e

Les tirages de ces deux étapes peuvent se faire selon une distribution uniforme, mais ce choix n'est pas pas optimum (cf. section 4.2.2 et chapitre 5).

4.2.2 Distribution de probabilités et optimisation de la qualité de test

Une fois l'ensemble d'éléments à couvrir connu, il faut déterminer une distribution pour le tirage des éléments. Si l'ensemble $E_C(D)$ est un ensemble de chemins, le cas est simple : nous utilisons la distribution uniforme sur les chemins.

Dans le cas contraire, il faut construire une distribution adéquate pour maximiser la probabilité p_{min}.

Cette section décrit comment obtenir une distribution dans les cas où l'ensemble $E_C(D)$ n'est pas un ensemble de chemins.

Étant donné $E_C(D) = \{e_1, e_2, ..., e_m\}$ avec $m > 0$, pour tout entier i et j dans $[1..m]$, pour tout $n > 0$ donné, nous définissons :

- α_i le nombre de chemins de $\mathcal{P}_{\leq n}$ qui passent par l'élément e_i ;
- α_{ij} le nombre de chemins de $\mathcal{P}_{\leq n}$ qui passent par e_i et par e_j : Notons que $\alpha_{ii} = \alpha_i$ et que $\alpha_{ij} = \alpha_{ji}$.
- π_i la probabilité de choisir l'élément e_i durant la première étape du processus décrit en section 4.2.1.

Calculons maintenant pour chaque i, la probabilité p_i que l'élément e_i soit atteint par un chemin :

$$p_i = \pi_i + \sum_{j \in [1..i-1] \bigcup [i+1..m]} \pi_j \frac{\alpha_{ij}}{\alpha_j}$$

En effet, la probabilité de choisir l'élément e_i à l'étape 1 est par définition π_i et la probabilité d'atteindre e_i en tirant aléatoirement (et uniformément) un chemin qui passe par un autre élément e_j est $\pi_j \frac{\alpha_{ij}}{\alpha_j}$. L'équation précédente se simplifie en :

$$p_i = \sum_{j \in [1..m]} \pi_j \frac{\alpha_{ij}}{\alpha_j} \tag{4.3}$$

Nous devons maintenant déterminer les probabilités $\{\pi_1, \pi_2, ..., \pi_m\}$ telles que $\sum \pi_i = 1$ et telles que la probabilité minimale $p_{min} = min\{p_i, i \in [1..m]\}$ soit maximisée. Cela peut s'énoncer sous forme d'un programme linéaire :

$$\text{Maximiser } p_{min} \text{ sous les contraintes : } \begin{cases} \forall i \leq m,\ p_{min} \leq p_i \\ \pi_1 + \pi_2 + ... + \pi_m = 1 \end{cases}$$

où les p_i sont calculés par l'équation 4.3. Des méthodes de résolution, comme celles de la famille des points intérieurs [75, 76], permettent de trouver une solution en temps polynomial par rapport à m.

Utilisons maintenant ce programme linéaire sur notre exemple. Étant donné le critère de couverture "tous les arcs" et une longueur maximale de $n = 9$, la table 4.3 représente les coefficients α_{ij} où i et j dénotent les lettres a à k. Par exemple, la valeur 6 de la ligne 3 et de la colonne 7 représente $\alpha_{cg} = 6$ c'est-à-dire le nombre de chemins complets de longueur inférieure ou égale à 9 qui passent à la fois par c et par g dans le graphe de la figure 4.2.

La table 4.3 des coefficients est calculée automatiquement en utilisant une fonction de dénombrement (cf. section 6.1.1 du chapitre 6) sur les structures combinatoires. Chaque α_{ij} est obtenu en construisant la structure combinatoire représentant l'ensemble des chemins du graphe qui passent par l'élément e_i et par l'élément e_j puis en utilisant une fonction de dénombrement (cf. section 6).

	a	b	c	d	e	f	g	h	i	j	k
a	9	0	9	0	5	7	5	5	6	6	3
b	0	5	3	5	1	2	1	4	3	3	2
c	9	3	12	3	6	9	6	8	9	8	4
d	0	5	3	5	1	2	1	4	3	3	2
e	5	1	6	1	6	3	6	3	5	5	1
f	7	2	9	2	3	9	3	7	7	5	4
g	5	1	6	1	6	3	6	3	5	5	1
h	5	4	8	4	3	7	3	9	7	7	2
i	6	3	9	3	5	7	5	7	9	6	3
j	6	3	8	3	5	5	5	7	6	9	0
k	3	2	4	2	1	4	1	2	3	0	5

Tab. 4.3 : *Table des α_{ij} correspondant à la description de la figure 4.2*

La table 4.4 présente le programme linéaire correspondant. Chaque ligne, excepté la dernière, est une inéquation qui correspond à une ligne du tableau 4.3. Le terme de gauche est la valeur à maximiser soit p_{min}. Le terme de droite est une des probabilités p_i calculées par l'équation 4.3. La première ligne, par exemple, exprime le fait que p_{min} est inférieur ou égal à la probabilité p_a d'atteindre l'arc a avec un chemin tiré aléatoirement. En maximisant p_{min}, on maximise la plus petite probabilité p_i et on améliore ainsi la qualité de test. La dernière ligne exprime seulement que la somme des probabilités π_i que nous recherchons doit être égale à 1.

La résolution de ce programme linéaire avec comme objectif de maximiser p_{min} peut se faire à l'aide de n'importe quelle méthode de résolution linéaire. Le logiciel MuPAD [109] qui nous permet de résoudre automatiquement ce système utilise la méthode du simplexe[3]. Cette méthode nous permet d'obtenir le système suivant :

$$\begin{cases} p_{min} = \frac{1}{2} \\ \pi_a = \pi_c = \pi_d = \pi_f = \pi_g = \pi_h = \pi_i = \pi_j = 0 \\ \pi_b = \pi_k = \frac{5}{16} \\ \pi_e = \frac{3}{8} \end{cases}$$

D'après la formule 4.2, la qualité de test optimisée pour N tests est alors de $1 - \frac{1}{2^N}$.

Le tableau 4.5 exprime numériquement la relation entre q_N et N. La qualité de test pour les valeurs de N que sont 125, 94, 63 et 32 est très très proche de 1.

[3]Non polynomiale mais efficace en pratique, la méthode du simplexe proposée par Dantzig (1947) est la plus répandue des méthodes de résolution de programmes linéaires.

$$
\begin{array}{lccccccccccc}
p_{min} \leq & \pi_a & & +\frac{3}{4}\pi_c & & +\frac{5}{6}\pi_e & +\frac{7}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{5}{9}\pi_h & +\frac{2}{3}\pi_i & +\frac{2}{3}\pi_j & +\frac{3}{5}\pi_k \\
p_{min} \leq & & \pi_b & +\frac{1}{4}\pi_c & +\pi_d & +\frac{1}{6}\pi_e & +\frac{2}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{4}{9}\pi_h & +\frac{1}{3}\pi_i & +\frac{1}{3}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \pi_a & +\frac{3}{5}\pi_b & +\pi_c & +\frac{3}{5}\pi_d & +\pi_e & +\pi_f & +\pi_g & +\frac{8}{9}\pi_h & +\pi_i & +\frac{8}{9}\pi_j & +\frac{4}{5}\pi_k \\
p_{min} \leq & & \pi_b & +\frac{1}{4}\pi_c & +\pi_d & +\frac{1}{6}\pi_e & +\frac{2}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{4}{9}\pi_h & +\frac{1}{3}\pi_i & +\frac{1}{3}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{1}{5}\pi_b & +\frac{1}{2}\pi_c & +\frac{1}{5}\pi_d & +\pi_e & +\frac{1}{3}\pi_f & +\pi_g & +\frac{1}{3}\pi_h & +\frac{5}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{1}{5}\pi_k \\
p_{min} \leq & \frac{7}{9}\pi_a & +\frac{2}{5}\pi_b & +\frac{3}{4}\pi_c & +\frac{2}{5}\pi_d & +\frac{1}{2}\pi_e & +\pi_f & +\frac{1}{2}\pi_g & +\frac{7}{9}\pi_h & +\frac{7}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{4}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{1}{5}\pi_b & +\frac{1}{2}\pi_c & +\frac{1}{5}\pi_d & +\pi_e & +\frac{1}{3}\pi_f & +\pi_g & +\frac{1}{3}\pi_h & +\frac{5}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{1}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{4}{5}\pi_b & +\frac{2}{3}\pi_c & +\frac{4}{5}\pi_d & +\frac{1}{2}\pi_e & +\frac{7}{9}\pi_f & +\frac{1}{2}\pi_g & +\pi_h & +\frac{7}{9}\pi_i & +\frac{7}{9}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \frac{2}{3}\pi_a & +\frac{3}{5}\pi_b & +\frac{3}{4}\pi_c & +\frac{3}{5}\pi_d & +\frac{5}{6}\pi_e & +\frac{7}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{7}{9}\pi_h & +\pi_i & +\frac{2}{3}\pi_j & +\frac{3}{5}\pi_k \\
p_{min} \leq & \frac{2}{3}\pi_a & +\frac{3}{5}\pi_b & +\frac{2}{3}\pi_c & +\frac{3}{5}\pi_d & +\frac{5}{6}\pi_e & +\frac{5}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{7}{9}\pi_h & +\frac{2}{3}\pi_i & +\pi_j & \\
p_{min} \leq & \frac{1}{3}\pi_a & +\frac{2}{5}\pi_b & +\frac{1}{3}\pi_c & +\frac{2}{5}\pi_d & +\frac{1}{6}\pi_e & +\frac{4}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{2}{9}\pi_h & +\frac{1}{3}\pi_i & & +\pi_k \\
1 = & \pi_a & +\pi_b & +\pi_c & +\pi_d & +\pi_e & +\pi_f & +\pi_g & +\pi_h & +\pi_i & +\pi_j & +\pi_k
\end{array}
$$

Tab. 4.4 : *Le programme linéaire correspondant au tableau 4.3*

q_N	1-2.10^{-38}	1-5.10^{-19}	1-10^{-19}	1-2.10^{-10}	0.9999	0.999	0.99	0.9
N	125	94	63	32	14	10	7	4

Tab. 4.5 : *Relation entre q_N et N pour le critère "tous les arcs" et une longueur de chemins inférieure ou égale à 9*

Remarquons que dans notre exemple, pour couvrir le critère "tous les arcs" avec une qualité de test donnée, il faut environ 9 fois moins de tests que pour couvrir le critère "tous les chemins de longueur inférieure ou égale à 9" avec la même qualité.

4.2.3 Couverture des chemins

Si $E_C(D)$ est un ensemble de chemins, nous avons vu en section 4.2.1 que tous les chemins permettant de couvrir cet ensemble ont une probabilité non nulle d'être tirés puisqu'un tirage aléatoire et uniforme est effectué parmi les chemins de $E_C(D)$.

Par contre, dans les cas où $E_C(D)$ n'est pas un ensemble de chemins, il peut exister des chemins dont la probabilité d'être tirés soit nulle puisque la distribution que nous construisons ne prend pas en compte cette contrainte. C'est le cas, par exemple, si l'on considère le graphe de la figure 4.3 (en se limitant aux chemins de longueur inférieure ou égale à 9) et le critère "tous les arcs". La résolution du

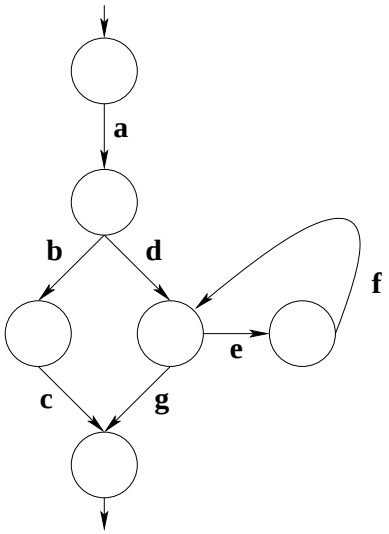


Fig. 4.3 : *Un graphe*

programme linéaire suivant :

$$\begin{cases} p_{min} \leq \pi_a + \pi_b + \pi_c + \pi_d + \pi_e + \pi_f + \pi_g \\ p_{min} \leq \frac{1}{5}\pi_a + \pi_b + \pi_c \\ p_{min} \leq \frac{1}{5}\pi_a + \pi_b + \pi_c \\ p_{min} \leq \frac{4}{5}\pi_a + \pi_d + \pi_e + \pi_f + \pi_g \\ p_{min} \leq \frac{3}{5}\pi_a + \frac{3}{4}\pi_d + \pi_e + \pi_f + \frac{3}{4}\pi_g \\ p_{min} \leq \frac{3}{5}\pi_a + \frac{3}{4}\pi_d + \pi_e + \pi_f + \frac{3}{4}\pi_g \\ p_{min} \leq \frac{4}{5}\pi_a + \pi_d + \pi_e + \pi_f + \pi_g \\ 1 = \pi_a + \pi_b + \pi_c + \pi_d + \pi_e + \pi_f + \pi_g \end{cases}$$

nous donne ce système comme solution :

$$\begin{cases} p_{min} = \frac{1}{2} \\ \pi_a = \pi_c = \pi_d = \pi_f = \pi_g = 0 \\ \pi_b = \frac{1}{2} \\ \pi_e = \frac{1}{2} \end{cases}$$

Avec cette distribution sur les éléments, le chemin `adg` ne peut pas être tiré. La résolution de ce problème dépend du nombre et des caractéristiques des chemins écartés ainsi que du nombre de tests considéré. Par exemple, si les chemins écartés ne couvrent aucun élément de $E_C(D)$ ou bien s'il s'agit de chemins infaisables (cf. section 4.4), on peut éventuellement les ignorer. Ou encore, si leur nombre k est négligeable par rapport au nombre N de tests à effectuer, on peut considérer

ces k chemins comme ayant été tirés puis tirer les $N-k$ chemins suivants en fonction de la distribution calculée.

Une autre solution, pour éviter ces probabilités nulles, est de choisir un élément λ de $E_C(D)$ tel que tous les chemins du graphe passent par cet élément et lui attribuer une probabilité π_λ (à déterminer) non nulle. Dans notre exemple, un tel élément serait a. Ensuite, il suffit de rajouter au programme linéaire l'équation $\pi_\lambda \geq v$ où v est la valeur choisie pour π_λ. La probabilité π_λ est à déterminer avec précaution : en effet, plus π_λ sera grand plus la qualité de test diminuera. Sur notre exemple, le programme linéaire devient :

$$\begin{cases} p_{min} \leq \pi_a + \pi_b + \pi_c + \pi_d + \pi_e + \pi_f + \pi_g \\ p_{min} \leq \frac{1}{5}\pi_a + \pi_b + \pi_c \\ p_{min} \leq \frac{1}{5}\pi_a + \pi_b + \pi_c \\ p_{min} \leq \frac{4}{5}\pi_a + \pi_d + \pi_e + \pi_f + \pi_g \\ p_{min} \leq \frac{3}{5}\pi_a + \frac{3}{4}\pi_d + \pi_e + \pi_f + \frac{3}{4}\pi_g \\ p_{min} \leq \frac{3}{5}\pi_a + \frac{3}{4}\pi_d + \pi_e + \pi_f + \frac{3}{4}\pi_g \\ p_{min} \leq \frac{4}{5}\pi_a + \pi_d + \pi_e + \pi_f + \pi_g \\ \pi_a = v \\ 1 = \pi_a + \pi_b + \pi_c + \pi_d + \pi_e + \pi_f + \pi_g \end{cases}$$

Le tableau 4.6 donne les différents p_{min} obtenus en fonction de quelques valeurs choisies pour v.

p_{min}	$\frac{1}{5}$	$\frac{9}{20}$	$\frac{17}{35}$	$\frac{37}{75}$	$\frac{49999}{100000}$
v	1	$\frac{1}{2}$	$\frac{1}{7}$	$\frac{1}{15}$	$\frac{1}{10000}$

Tab. 4.6 : *Relation entre q et v*

Si aucun élément de $E_C(D)$ ne convient, on ajoutera un nouvel élément arbitraire pour jouer ce rôle et si cet élément est tiré alors un tirage parmi tous les chemins considérés du graphe sera effectué. Dans ce cas, le programme linéaire sera complètement reconstruit en fonction de cet élément λ. Chaque inéquation de la forme $p \leq p_e$ où e est un élément de $E_C(D)$ sera modifié de la façon suivante $p \leq p_e + \frac{\alpha_e}{\alpha_\lambda}\pi_\lambda$ où α_λ est le nombre de chemins considérés du graphe. L'équation $1 = \sum_{e \in E_C(D)} \pi_e$ sera complétée par $1 = \sum_{e \in E_C(D)} \pi_e + \pi_\lambda$.

4.3 Des chemins aux données d'entrée

Une fois la distribution déterminée, le tirage des chemins peut s'effectuer. Puis, comme avec toute méthode de test basée sur des chemins, on doit calculer un ensemble de données d'entrée qui permettent de passer par chacun de ces

chemins. Le passage des chemins aux données d'entrée est spécifique à chaque description. Lorsque cette dernière est un simple automate fini (sans condition de branchement d'une transition), le passage est immédiat.

Mais dans le cas de descriptions plus compliquées, c'est-à-dire avec des types de données et/ou des conditions sur certaines transitions comme c'est le cas pour les graphes de contrôle ou les statecharts, on a vu au chapitre 1 que cela pouvait être plus difficile : il existe des cas où c'est décidable et des cas où cela ne l'est pas.

4.4 Influence des chemins infaisables sur la qualité de test

Si aucune donnée d'entrée ne permet de passer par un chemin sélectionné, par définition le chemin est un chemin infaisable (cf. chapitre 1). La présence des chemins infaisables affecte notre approche différemment selon que l'ensemble des éléments à couvrir est un ensemble de chemins ou non.

Si l'ensemble $E_C(D)$ est un ensemble de chemins, la présence de chemins infaisables n'introduit pas de biais sur la distribution obtenue sur les chemins faisables. Celle-ci est également uniforme. En effet, comme nous l'avons déjà vu en section 3.1.2, un tirage uniforme dans un ensemble F (ensemble de chemins faisables et infaisables) permet de faire du tirage uniforme sur un sous-ensemble E (chemins faisables) de F.

Par contre, l'influence des chemins infaisables sur la distribution construite pour couvrir les éléments qui ne sont pas des chemins d'un ensemble $E_C(D)$ peut être importante. En effet, cette distribution est optimisée dans les cas où tous les chemins sont faisables mais ne l'est plus dans le cas contraire.

Reprenons le graphe de la figure 4.2 et le cas du critère de couverture "tous les arcs", et supposons que tous les chemins passant à la fois par l'arc b et l'arc c du graphe de la figure 4.2 soient infaisables. Dans ce cas, avec la distribution calculée en section 4.2.2, la probabilité minimum d'atteindre un élément lors de l'exécution d'un test diminue. En effet, si l'on calcule l'ensemble des probabilités en tenant compte des 3 chemins infaisables qui sont `bdhicfk`, `bdhicegj` et `bdhicfhj`, on obtient :

$$\begin{cases} p(a) = p(c) = \frac{5}{16} \times \frac{3}{4} + \frac{3}{8} = \frac{39}{64} \\ p(b) = p(d) = \frac{5}{16} \times \frac{1}{4} + \frac{5}{16} = \frac{25}{64} \\ p(e) = p(g) = \frac{5}{16} \times \frac{1}{4} + \frac{3}{8} = \frac{29}{64} \\ p(f) = \frac{5}{16} \times \frac{3}{4} + \frac{3}{8} \times \frac{3}{5} = \frac{147}{320} \\ p(h) = \frac{5}{16} \times \frac{1}{4} + \frac{5}{16} \times \frac{1}{2} + \frac{3}{8} \times \frac{2}{5} = \frac{123}{320} \\ p(i) = p(j) = \frac{5}{16} \times \frac{2}{4} + \frac{3}{8} \times \frac{4}{5} = \frac{73}{160} \end{cases}$$

Ce qui nous donne un p_{min} de $\frac{123}{320}$ soit 0.38, ce qui est nettement inférieur au p_{min}

calculé qui était de $\frac{1}{2}$.

Un moyen de prendre en considération ces chemins infaisables est de rendre plus précise la spécification combinatoire afin qu'elle décrive le plus précisément possible les chemins faisables. Une première application de cette idée a été utilisée lors de nos expériences (cf section 7.5.1).

4.5 Conclusion

Nous proposons une nouvelle méthode de test statistique automatisable et qui s'adapte à priori à tout type de description pourvu qu'il soit modélisable sous forme de graphe. La figure 4.4 présente un schéma général en trois étapes principales pour la conception d'outils : une phase d'analyse qui permet de construire les spécifications combinatoires à partir de la description D et du critère C considéré, une phase de tirage des N chemins de longueur inférieure ou égale à n et une phase de résolution de contraintes pour passer des chemins tirés aux données de test. Dans les cas où la phase de résolution échoue, on retourne à la phase de tirage.

Suivant ce schéma, nous avons développé un outil (cf. chapitre 6) qui permet une première automatisation du test statistique structurel en appliquant notre méthode aux travaux de Thévenod-Fosse et Waeselynck [111] qui ont inspiré notre approche.

Si notre méthode est générale pour la sélection des chemins, elle ne peut l'être pour le passage des chemins aux données d'entrée. En effet, les prédicats calculables à partir de ces chemins diffèrent syntaxiquement et sémantiquement d'une description à l'autre. Par conséquent, à chaque méthode de description, il faut un module spécifique pour trouver les entrées qui permettent d'exécuter ces chemins.

Concernant le calcul de la qualité de test, il nous reste à résoudre trois points délicats. Les deux premiers concernent les chemins : ceux qui sont écartés par le calcul de la distribution sur les éléments de $E_C(D)$ (section 4.2.3) et que l'on voudrait quand même pouvoir tirer, et ceux qui sont infaisables (section 4.4) et qui faussent l'optimisation de la distribution sur les éléments de $E_C(D)$. Le troisième point porte sur l'exécution indépendante des tests. Le calcul de la qualité de test tel qu'il a été présenté est correct lorsque chaque test est exécuté indépendamment du précédent, c'est-à-dire que les tests précédents n'ont aucune influence sur les tests en cours. Si ce n'est pas le cas, alors le calcul de la probabilité d'exécution des éléments (chemins ou non) n'est plus juste. L'exécution non indépendante des tests qui est parfois inévitable dans des situations réelles, comme nous le verrons dans le chapitre 7, peut s'avérer aussi très gênante car elle peut perturber les expériences et provoquer une divergence avec les résultats théoriques.

Ces différents points demandent donc à être étudiés.

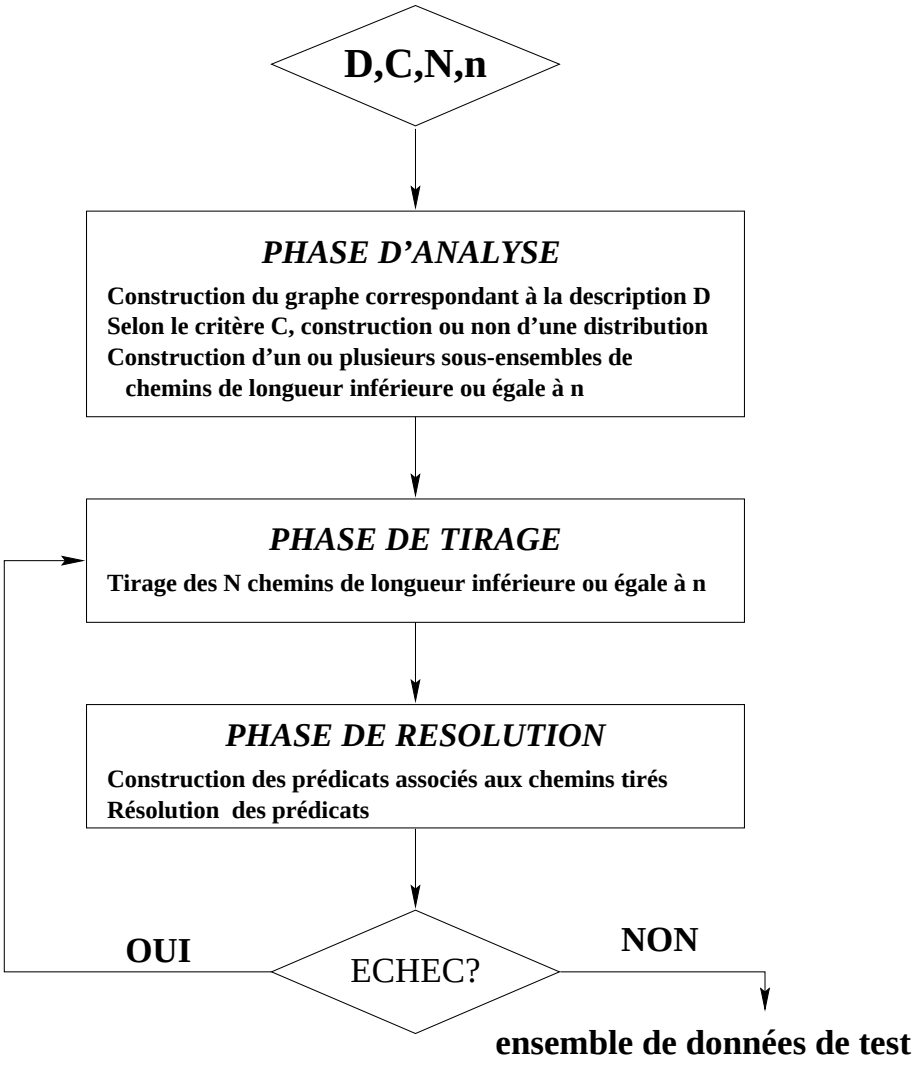


Fig. 4.4 : *Schéma général des prototypes implémentant notre approche*

Troisième partie

Première application : le test statistique structurel

Chapitre 5

Application au test statistique structurel

Ce chapitre présente une application de notre méthode au test statistique structurel. Historiquement, nous avons commencé nos travaux par cette application afin de nous convaincre de l'applicabilité des techniques combinatoires au test de logiciel. Le choix du test statistique structurel a été guidé par les travaux de Thévenod-Fosse et Waeselynck [112], et en particulier par la possibilité de récupérer les fichiers d'expériences qu'elles avaient utilisés pour évaluer leur méthode. Ainsi nous avons pu réaliser de premières expériences et une première évaluation de notre approche.

Nous rappelons que l'approche que nous proposons est composée des deux grandes étapes suivantes :

1. On effectue un tirage parmi les chemins de longueur inférieure ou égale à n du graphe de contrôle selon une distribution uniforme en utilisant des outils de génération de structures combinatoires (section 6.1, [33])
2. On résout les prédicats associés à ces chemins à l'aide d'un solveur de contraintes (page 26).

Dans un premier temps, nous détaillons la première étape appliquée au test statistique structurel. Nous présentons également les premières approches développées pour construire la distribution dans les cas où l'ensemble $E_C(D)$ n'est pas un ensemble de chemins. En effet, avant d'obtenir la version utilisant la programmation linéaire, nous avons étudié la distribution uniforme des éléments du critère à couvrir, comme extension naturelle de la distribution uniforme sur les chemins, puis une distribution basée sur la notion de dominant dont le but était de favoriser les éléments dont la probabilité d'être atteint était la plus faible. Cette dernière distribution est celle qui a été utilisée pour effectuer les expériences du chapitre 7.

5.1 Génération de chemins

Comme nous l'avons vu en section 1.4, le test statistique structurel est une méthode de test qui combine le test statistique, pour la partie génération des données d'entrée et le test structurel, pour la partie représentation du programme. Par conséquent, le type de description utilisé est le graphe de contrôle du programme et les critères de couverture considérés, ceux du test structurel.

Étant donné un critère de couverture C et un nombre de test N, notre objectif est d'optimiser la qualité de test $q_{C,N}$. Selon la nature des éléments du critère, c'est-à-dire s'ils sont des chemins ou non, nous avons vu au chapitre précédent que le procédé n'était pas le même.

En effet, si les éléments du critère ne sont pas des chemins, alors le tirage des chemins (première étape de notre approche) s'effectue en deux étapes :

1. choisir aléatoirement un élément e de $E_C(D)$, selon une distribution adéquate des probabilités.
2. tirer uniformément un chemin de longueur inférieure ou égale à n passant par e.

La première étape est celle dont l'étude nous a donné le plus de mal. Avant d'arriver à la distribution optimisée décrite au chapitre 4, plusieurs approches ont été étudiées. Dans la section suivante, nous présentons les deux plus importantes qui sont :

- la distribution uniforme : l'idée étant que puisque le tirage est uniforme parmi les chemins, utilisons également une distribution uniforme sur les éléments.
- la distribution basée sur les "dominances" : l'idée était de favoriser le tirage d'éléments de faible probabilité par rapport à des éléments plus probables. Cette distribution est celle qui a été implantée dans notre prototype pour faire nos premières expériences.

5.2 Déterminer la distribution

5.2.1 Tirage uniforme parmi les éléments d'un critère

Pour N tests, l'algorithme est le suivant :

1. Tirer de manière aléatoire uniforme N éléments $e_1, ..., e_N$ dans E_C
2. Pour chaque élément e_i de $\{e_1, ..., e_N\}$, tirer de manière aléatoire uniforme un chemin dans l'ensemble des chemins de longueur inférieure ou égale à n passant par e_i.

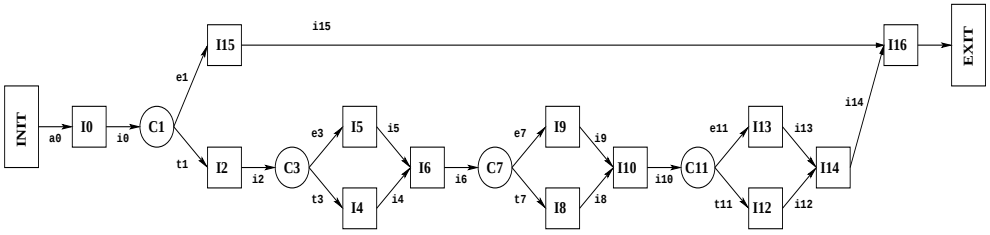


Fig. 5.1 : *Graphe de contrôle du programme FCT2*

Exemple 20 (FCT2 et le critère "toutes les instructions")
D'après la figure 5.1, l'ensemble E_C des éléments du critère "toutes les instructions" est le suivant : $E_C = \{I0, I2, I4, I5, I6, I8, I9, I10, I12, I13, I14, I15, I16\}$
Il y a 13 éléments c'est-à-dire 13 blocs d'instructions et 9 chemins.
Les probabilités d'atteindre chaque élément du critère en exécutant un *jeu de test sélectionné par notre méthode sont les suivantes :*

- $p(I0) = p(I16) = 1$
- $p(I2) =$

  $\frac{1}{13}$ *cas où on choisit I2 à l'étape 1*

  $+0$ *cas où on choisit I15 à l'étape 1 (on ne peut pas atteindre I2)*

  $+2 \times \frac{1}{13} \times \frac{8}{9}$ *cas où l'on choisit I0 ou I16 à l'étape 1 puis un chemin passant par I2 et I0 ou I16*

  $+9 \times \frac{1}{13}$ *dans les autres cas car on est toujours sûr de passer par I2*

  Ce qui donne $p(I2) = \frac{106}{117}$.
  En appliquant le même raisonnement, on obtient également :
  $p(I6) = p(I10) = p(I14) = \frac{106}{117}$.
- $p(I4) =$

  $\frac{1}{13}$ *cas où on choisit I4 à l'étape 1*

  $+0$ *cas où on choisit I5 ou I15 à l'étape 1 (on ne peut pas atteindre I4)*

  $+2 \times \frac{1}{13} \times \frac{4}{9}$ *cas où l'on choisit I0 ou I16 à l'étape 1*

  $+4 \times \frac{1}{13} \times \frac{4}{8}$ *cas où l'on choisit I2,I6,I10 ou I14 à l'étape 1*

  $+4 \times \frac{1}{13} \times \frac{2}{4}$ *dans les autres cas*

  Ce qui donne $p(I4) = \frac{53}{117}$.
  En appliquant un raisonnement similaire, on obtient également :
  $p(I5) = p(I8) = p(I9) = p(I12) = p(I13) = \frac{53}{117}$.
- $p(I15) = \frac{1}{13} + 2 \times \frac{1}{13} \times \frac{1}{9} = \frac{11}{117}$

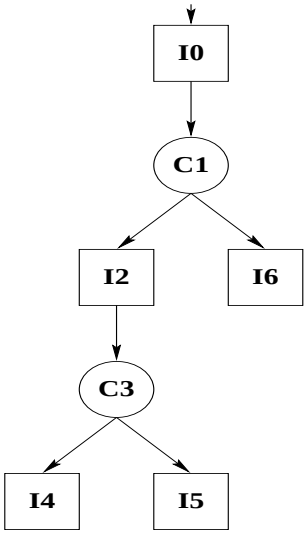
On remarque donc que $p_{min} = p(I5) = \frac{53}{117}$. *Pour obtenir une qualité de test* $q_N = 0.9999$, *il faudrait faire* $N \geq \frac{\log(1-0.9999)}{\log(1-p_{min})}$ *tests soit* $N \geq 94$ *tests.*

Une distribution uniforme sur l'ensemble E_C, bien que naturelle, ne permet pas toujours de maximiser les probabilités les plus faibles. En effet, il existe des éléments peu ou non contraints : quel que soit le chemin suivi, ils sont très souvent voir toujours atteints.

Exemple 21 (C= "toutes les instructions")
Dans le cas ci-contre, $E_C = \{I0, I2, I4, I5, I6\}$ *et* $p_{min} = p(I6) = \frac{4}{15}$.
Si au lieu d'utiliser une distribution uniforme, nous utilisons la distribution suivante :

$$\begin{cases} p(I0) = p(I2) = 0 \\ p(I4) = p(I5) = p(I6) = \frac{1}{3} \\ p_{min} = p(I4) = \frac{1}{3} \end{cases}$$

Ce qui revient à tirer uniformément parmi les éléments de l'ensemble $S = \{I4, I5, I6\}$. *Alors on obtient un meilleur* p_{min}. *Cette nouvelle distribution est optimisée pour cet exemple.*



Dans l'exemple ci-dessus, on peut remarquer que la génération aléatoire et uniforme de chemins de longueur inférieure ou égale à n, avec $n \geq 4$, aurait donné de meilleurs résultats puisque la probabilité d'atteindre un élément est de $\frac{1}{3}$. Le choix du critère de couverture par l'utilisateur est donc important et en particulier ici, où l'ensemble de chemins (même sans limitation de longueur) est fini (et petit).

Nos observations sur la distribution uniforme et les cas sur lesquels elle n'était pas optimale, nous ont amené à construire une distribution qui prenne en considération les particularités des éléments.

5.2.2 Construire la distribution à l'aide des dominances

Cette approche est celle qui a été implantée dans le prototype AuGuSTe pour effectuer les expériences. C'est pourquoi elle est présentée ici. Si le tirage s'effectue de manière uniforme dans E_C, alors la probabilité d'un élément e_i d'être atteint par un test est (cf. section 4.2.2) :

$$p_i = \sum_{j \in [1..m]} \frac{1}{|E_C|} \frac{\alpha_{ij}}{\alpha_j}$$

La probabilité d'un élément étant fortement liée à la cardinalité de E_C, une façon de l'augmenter, et donc d'espérer augmenter p_{min}, est de faire diminuer la

cardinalité de E_C c'est-à-dire choisir un sous-ensemble pertinent S de E_C dans lequel un tirage uniforme sera effectué. La probabilité d'un élément s'exprimerait alors de la façon suivante :

$$p_i = \sum_{j \in S} \frac{1}{|S|} \frac{\alpha_{ij}}{\alpha_j}$$

Ce sous-ensemble doit assurer la couverture de tous les éléments du critère, sans écarter de chemins. Une première idée serait de le réduire à un élément mais dans ce cas, cela reviendrait au critère "tous les chemins". La méthode implémentée dans le prototype se base sur les notions de classes d'équivalence d'éléments et de dominance d'éléments par rapport à d'autres.

5.2.2.1 Définitions

Afin de partitionner les éléments selon l'ensemble des chemins qui passent par eux, nous définissons la notion d'équivalence entre éléments de même critère de la façon suivante :

Définition 12 (Équivalence d'éléments) *Deux éléments e_i et e_j sont équivalents si tout chemin complet passe soit par e_i et e_j soit par aucun des deux.*

Dans le graphe de la figure 5.1, les sommets $I2$, $I6$, $I10$ et $I14$ sont équivalents alors que les arcs $e3$ et $t3$ ne le sont pas.

Ensuite, nous aimerions ordonner les classes d'équivalence entre éléments selon la relation d'inclusion appliquée à l'ensemble des chemins qui passent par les éléments de la classe. Cet ordre n'est rien d'autre qu'une variante de la notion de dominance [9] où les sommets du graphe sont remplacés par les classes d'équivalence.

Définition 13 (Dominance entre sommets [9]) *Un sommet A d'un graphe de contrôle* **domine** *le sommet B, ce qu'on écrit A dom B, si tout chemin partant du sommet initial et arrivant à B passe par A. Un sommet se domine lui-même et l'entrée d'une boucle domine tous les sommets de la boucle.*

Cette définition s'étend facilement aux arcs d'un graphe de contrôle.

Définition 14 (Dominance entre classes d'équivalence)
Une classe d'équivalence A d'éléments d'un graphe de contrôle **domine** *une classe d'équivalence B si tous les chemins complets qui passent par un élément de B passent par tous les éléments de A. Si $\mathcal{C}_A$ (resp. $\mathcal{C}_B$) est l'ensemble des chemins complets passant par les éléments de la classe d'équivalence A (resp. B) alors on a :*

$$\mathcal{C}_B \subset \mathcal{C}_A$$

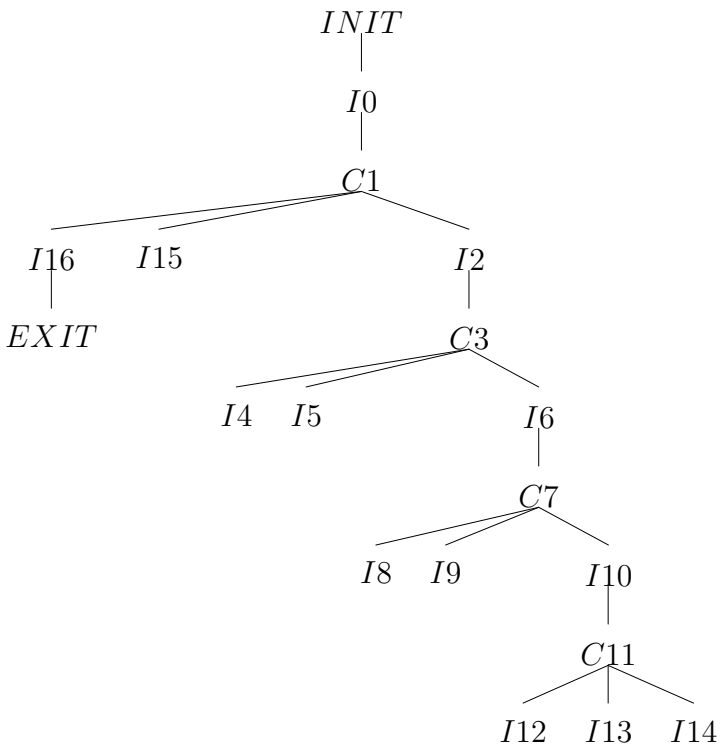


Fig. 5.2 : *Arbre des dominants des sommets du graphe de contrôle de FCT2*

On représente habituellement l'information sur les dominants sous forme d'un arbre, appelé *arbre des dominants*, qui a comme racine le sommet initial et où chaque noeud domine uniquement ses descendants. La figure 5.2 représente l'arbre des dominants du graphe de contrôle du programme FCT2.

La figure 5.3 représente l'arbre des dominants où chaque sommet a été remplacé par une classe d'équivalence.

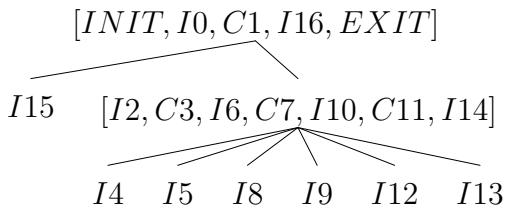


Fig. 5.3 : *Arbre des dominants des classes d'équivalence d'éléments du graphe de contrôle du programme FCT2*

On peut remarquer que les feuilles de l'arbre des dominants de la figure 5.2 appartiennent toutes à des classes d'équivalences différentes. Les deux feuilles

$EXIT$ et $I14$ qui ont disparu entre les deux versions de l'arbre des dominants appartiennent à des classes d'équivalence qui sont des noeuds internes de l'arbre comme l'étaient les autres éléments de la classe.

5.2.2.2 Méthode

Pour construire une distribution à l'aide des dominances, on choisit un sous-ensemble S de E_C qui contient au moins un représentant de chaque classe qui se trouvent aux feuilles du graphe des dominants[1]. Cette distribution donne :

- à tous les éléments de S une probabilité $\frac{1}{|S|}$ d'être tiré à l'étape 1,
- aux autres éléments une probabilité nulle.

Pour le programme FCT2, cela donne par exemple :

$$S = \{I4, I5, I8, I9, I12, I13, I15\}$$

En effet, les classes qui se trouvent aux feuilles sont telles que leurs éléments ont les probabilités les plus faibles d'être atteints.

Il peut arriver que cet ensemble écarte des chemins. La présence de boucles ou d'instructions conditionnelles implique une attention particulière. En effet, si l'on considère l'exemple de la figure 5.4 et le critère "toutes les instructions", il y a 2 classes d'équivalence $[I0, C1, I3]$ et $[I2]$. Le graphe des dominants a pour racine la classe $[I0, C1, I3]$ et comme unique feuille la classe $[I2]$. L'ensemble S est réduit alors à l'élément $I2$ et le chemin `ad` n'est pas plus tirable.

Pour pallier ce défaut, nous avons ajouté une règle, décrite en section 4.2.3, à cette méthode :

1. Construire un ensemble S qui contient un représentant des classes qui sont des feuilles de l'arbre des dominants
2. Si cet ensemble S ne permet pas d'atteindre tous les chemins, alors le modifier en rajoutant un représentant de la classe qui est à la racine de l'arbre des dominants.

La racine étant la classe contenant le sommet initial du graphe de contrôle, on est sûr de n'écarter a priori aucun chemin.

Dans le cas de l'exemple 21 (section 5.2.1), on obtient immédiatement l'ensemble qui permet d'avoir le meilleur p_{min} :

$$S' = \{I4, I5, I6\}$$

Cette approche, bien que plus efficace que la distribution uniforme, ne permet pas toujours de maximiser les probabilités les plus faibles comme nous le montre l'exemple suivant.

[1] La représentation associée à la relation d'inclusion d'ensembles de chemins peut ne pas être un arbre, cf. l'exemple de [8]

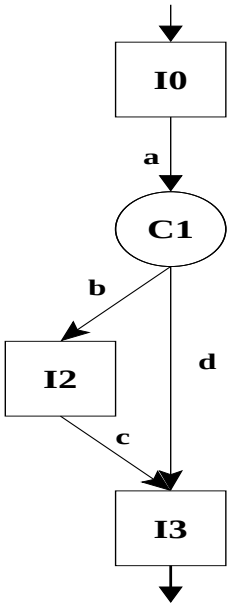


Fig. 5.4 :

Exemple 22 (FCT2 et le critère "toutes les instructions")
Dans le cas du programme FCT2, l'ensemble S construit à l'aide de cette méthode est

$$S = \{I4, I5, I8, I9, I12, I13, I15\}$$

Ce qui nous donne les probabilités suivantes :

$$\begin{cases} p(I0) = p(I16) = 1 \\ p(I2) = p(I6) = p(I10) = p(I14) = \frac{6}{7} \\ p(I4) = p(I5) = p(I8) = p(I9) = p(I12) = p(I13) = \frac{1}{7} + 4 \times \frac{1}{7} \times \frac{2}{4} = \frac{3}{7} \\ p_{min} = p(I15) = \frac{1}{7} \end{cases}$$

Pour obtenir une qualité de test $q_N = 0.9999$, il faudrait faire $N \geq \frac{\log(1-0.9999)}{\log(1-p_{min})}$ soit $N \geq 60$ tests.

Si l'on prend maintenant l'ensemble $S' = \{I15, I4, I5\}$ qui permet de couvrir toutes les instructions sans écarter de chemin, on obtient les probabilités suivantes :

$$\begin{cases} p(I0) = p(I16) = 1 \\ p(I2) = p(I6) = p(I10) = p(I14) = \frac{2}{3} \\ p(I4) = p(I5) = p(I8) = p(I9) = p(I12) = p(I13) = \frac{1}{3} \\ p_{min} = p(I15) = \frac{1}{3} \end{cases}$$

Le p_{min} de S' est supérieur à celui de S, donc pour un nombre de tests N donné, la qualité de tests sera supérieure.

En fait, la construction du sous-ensemble S peut être vue comme la construction d'une distribution non-uniforme sur les éléments de E_C. Ainsi par exemple pour le programme FCT2, la distribution construite est la suivante :

$$\begin{cases} \pi_{I0} = \pi_{I16} = 0 \\ \pi_{I2} = \pi_{I6} = \pi_{I10} = \pi_{I14} = 0 \\ \pi_{I4} = \pi_{I5} = \pi_{I8} = \pi_{I9} = \pi_{I12} = \pi_{I13} = \frac{1}{7} \\ \pi_{I15} = \frac{1}{7} \end{cases}$$

Si l'on retire les contraintes liées à l'optimisation de la qualité de test et à la couverture de tous les chemins du graphe, la méthode utilisée pour trouver l'ensemble S est celle proposée par Agrawal [8] où les *super blocks* sont nos classes d'équivalence. Simplement, nous avons construit notre graphe de dominance directement à partir des ensembles de chemins plutôt que de calculer et de manipuler plusieurs graphes, comme l'arbre des pré dominants[2] ou celui des post-dominants[3].

Toujours sans les contraintes, cette méthode peut être vue comme une application, aux critères tous les enchaînements ou toutes les instructions, de la méthode de Bertolino et Marré [90] pour trouver l'ensemble S.

5.2.3 Comparaison par rapport à la solution optimisée

Dans le chapitre 4, nous avons présenté une solution optimisée pour construire l'ensemble S. Elle consiste à construire une distribution non uniforme sur E_C en résolvant le problème suivant :

$$\text{Maximiser } p_{min} \text{ sous les contraintes : } \begin{cases} \forall i,\ p_i = \sum_{j \in [1..m]} \pi_j \frac{\alpha_{ij}}{\alpha_j} \\ \forall i \leq m,\ p_{min} \leq p_i \\ \pi_1 + \pi_2 + ... + \pi_m = 1 \end{cases}$$

Pour utiliser cette méthode, il faut commencer par calculer les α_{ij}. Le tableau 5.1 récapitule les valeurs des α_{ij} calculées pour le programme FCT2 et le critère "toutes les instructions".

Le système linéaire correspondant au programme FCT2 et au critère "toutes les instructions" est représenté en figure 5.5. Les π_n représentent la probabilité de choisir In dans l'ensemble E_C.

[2]Un noeud i pré-domine un noeud j si tous les chemins qui vont du point d'entrée $INIT$ au noeud j passent par le noeud i.

[3]Un noeud i post-domine un noeud j si tous les chemins qui vont du noeud j au noeud de sortie $EXIT$ passent par le noeud i

	$I0$	$I2$	$I4$	$I5$	$I6$	$I8$	$I9$	$I10$	$I12$	$I13$	$I14$	$I15$	$I16$
$I0$	9	8	4	4	8	4	4	8	4	4	8	1	9
$I2$	8	8	4	4	8	4	4	8	4	4	8	0	8
$I4$	4	4	4	0	4	4	4	4	4	4	4	0	4
$I5$	4	4	0	4	4	4	4	4	4	4	4	0	4
$I6$	8	8	4	4	8	4	4	8	4	4	8	0	8
$I8$	4	4	4	4	4	4	0	4	4	4	4	0	4
$I9$	4	4	4	4	4	0	4	4	4	4	4	0	4
$I10$	8	8	4	4	8	4	4	8	4	4	8	0	8
$I12$	4	4	4	4	4	4	4	4	4	0	4	0	4
$I13$	4	4	4	4	4	4	4	4	0	4	4	0	4
$I14$	8	8	4	4	8	4	4	8	4	4	8	0	8
$I15$	1	0	0	0	0	0	0	0	0	0	0	1	1
$I16$	9	8	4	4	8	4	4	8	4	4	8	1	9

Tab. 5.1 : *Matrice des α_{ij} pour le programme FCT2 et le critère "toutes les instructions"*

$$
\begin{array}{lrrrrrrrrrrrrrr}
p_{min} \leq & \pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & +\pi_{15} & +\pi_{16} \\
p_{min} \leq & \frac{8}{9}\pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & & +\frac{8}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & +\pi_4 & & +\frac{1}{2}\pi_6 & +\frac{1}{2}\pi_8 & +\frac{1}{2}\pi_9 & +\frac{1}{2}\pi_{10} & +\frac{1}{2}\pi_{12} & +\frac{1}{2}\pi_{13} & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & & +\pi_5 & +\frac{1}{2}\pi_6 & +\frac{1}{2}\pi_8 & +\frac{1}{2}\pi_9 & +\frac{1}{2}\pi_{10} & +\frac{1}{2}\pi_{12} & +\frac{1}{2}\pi_{13} & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{8}{9}\pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & & +\frac{8}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & +\frac{1}{2}\pi_4 & +\frac{1}{2}\pi_5 & +\frac{1}{2}\pi_6 & +\pi_8 & & +\frac{1}{2}\pi_{10} & +\frac{1}{2}\pi_{12} & +\frac{1}{2}\pi_{13} & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & +\frac{1}{2}\pi_4 & +\frac{1}{2}\pi_5 & +\frac{1}{2}\pi_6 & & +\pi_9 & +\frac{1}{2}\pi_{10} & +\frac{1}{2}\pi_{12} & +\frac{1}{2}\pi_{13} & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{8}{9}\pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & & +\frac{8}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & +\frac{1}{2}\pi_4 & +\frac{1}{2}\pi_5 & +\frac{1}{2}\pi_6 & +\frac{1}{2}\pi_8 & +\frac{1}{2}\pi_9 & +\frac{1}{2}\pi_{10} & +\pi_{12} & & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{4}{9}\pi_0 & +\frac{1}{2}\pi_2 & +\frac{1}{2}\pi_4 & +\frac{1}{2}\pi_5 & +\frac{1}{2}\pi_6 & +\frac{1}{2}\pi_8 & +\frac{1}{2}\pi_9 & +\frac{1}{2}\pi_{10} & & +\pi_{13} & +\frac{1}{2}\pi_{14} & & +\frac{4}{9}\pi_{16} \\
p_{min} \leq & \frac{8}{9}\pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & & +\frac{8}{9}\pi_{16} \\
p_{min} \leq & \frac{1}{9}\pi_0 & & & & & & & & & & & +\pi_{15} & +\frac{1}{9}\pi_{16} \\
p_{min} \leq & \pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & +\pi_{15} & +\pi_{16} \\
1 = & \pi_0 & +\pi_2 & +\pi_4 & +\pi_5 & +\pi_6 & +\pi_8 & +\pi_9 & +\pi_{10} & +\pi_{12} & +\pi_{13} & +\pi_{14} & +\pi_{15} & +\pi_{16}
\end{array}
$$

Fig. 5.5 : *Le système linéaire à résoudre pour le programme FCT2 et le critère "toutes les instructions"*

La résolution de ce système se fait facilement à l'aide d'outils mathématiques comme MuPAD [109]. Une distribution [4] solution du système précédent est :

$$\begin{cases} \pi_2 = \pi_6 = \pi_{10} = \pi_{14} = 0 \\ \pi_4 = \pi_5 = \pi_8 = \pi_9 = \pi_{12} = \pi_{13} = 0 \\ \pi_{16} = 0 \\ \pi_0 = \frac{3}{4} \\ \pi_{15} = \frac{1}{4} \end{cases}$$

Ce qui nous donne les probabilités suivantes d'être atteint pour chaque élément du critère "toutes les instructions" :

$$\begin{cases} p(I0) = p(I16) = 1 \\ p(I2) = p(I6) = p(I10) = p(I14) = \frac{3}{4} \times \frac{8}{9} = \frac{2}{3} \\ p(I15) = \frac{1}{4} + \frac{3}{4} \times \frac{1}{9} = \frac{1}{3} \\ p(I4) = p(I5) = p(I8) = p(I9) = p(I12) = p(I13) = \frac{3}{4} \times \frac{4}{9} = \frac{1}{3} \end{cases}$$

Le p_{min} est alors de $\frac{1}{3}$. Pour une qualité de test de $q_N = 0.9999$, il nous faut ici $N \geq \frac{\log(1-0.9999)}{\log(1-\frac{1}{3})}$ tests soit au moins 23 tests. Ce qui est beaucoup mieux que ce qui est obtenu avec la méthode des dominants.

5.3 Des chemins aux données d'entrée

Une fois les chemins sélectionnés, il faut déterminer les entrées qui permettront l'exécution de ces chemins. Cette étape commune à toute méthode de test structurel se fait de la même manière que celle décrite en section 1.2.2. Tous les solveurs classiques utilisés par les autres méthodes de test structurel peuvent être réutilisés.

Dans notre étude, nous avons utilisé une adaptation du solveur de contraintes randomisé, développé dans l'outil GATeL [87] présenté au chapitre 2. Le lecteur trouvera en section 1.2.3.4 et 6.1.3 plus de détails concernant cette méthode de résolution.

Comme toutes les méthodes de test structurel, notre méthode doit savoir gérer la présence de chemins infaisables ou soupçonnés d'être infaisables. Au chapitre 2, nous avons vu que les méthodes de test existantes avaient différentes stratégies pour gérer ce type de problème.

Dans notre cas, nous n'avions pas de politique générale sur ces sujets. Une première stratégie a consisté à maintenir à jour un ensemble de chemins infaisables. Malheureusement, les programmes que nous avons traités contenaient tellement de chemins infaisables que nous avons eu des problèmes de mémoire.

[4] Il s'agit d'une solution parmi toutes celles qui permettent de résoudre le système linéaire.

L'approche actuelle consiste, pour chaque chemin infaisable ou soupçonnés d'être infaisables, à retirer un autre chemin jusqu'à n'obtenir que des chemins faisables.

5.4 Évaluation de cette méthode

Bien que la distribution utilisant la programmation linéaire soit la plus efficace sous certaines hypothèses, nous avons vu au chapitre précédent que la présence de chemins infaisables pouvait biaiser en pratique cette distribution. Il nous faut donc évaluer par des expériences, l'influence de ce phénomène. De plus, toute nouvelle méthode de tests se doit être évaluée par rapport aux méthodes qui existent déjà. Pour cela, nous avons développé un prototype AuGuSTe que nous présentons au chapitre 6.

Dans ce chapitre, nous avons présenté une nouvelle approche pour le test statistique structurel. Cette approche construit une distribution sur les chemins du graphe de contrôle et non sur les domaines des données d'entrées comme cela est le cas pour la méthode de Pascale Thévenod-Fosse et Hélène Waeselynck [111]. Une étude comparative basée sur les distributions est impossible. La comparaison a donc été effectuée expérimentalement en reprenant les expériences faites lors de l'évaluation de la méthode du LAAS. Les chapitres suivants présentent le prototype AuGuSTe, implémentant notre méthode, qui a été développé et les expériences effectuées et les résultats obtenus.

Chapitre 6

Le prototype AuGuSTe

À partir de l'approche présentée au chapitre 4, il est possible de développer toute une famille d'outils qui permettent d'implémenter des applications de notre approche à des méthodes de test, selon sur le schéma général proposé par la figure 4.4. Le prototype AuGuSTe (Automated Generation of Statistical Tests) est un exemple d'outil qui implémente l'application de notre approche au test statistique structurel présentée au chapitre 5. Il existe actuellement deux versions de cet outil : la première est celle qui nous a servi pour réaliser la majorité de nos expériences présentées au chapitre 7, et la seconde est celle qui sera utilisée dans la suite de nos travaux. La différence essentielle entre ces deux versions est la manière dont est construite la distribution sur les éléments pour les critères "tous les enchaînements" et "toutes les instructions". Une présentation de ces deux constructions a déjà été vue dans le chapitre 5.

AuGuSTe a une architecture modulaire afin de pouvoir facilement changer le langage des programmes, le solveur de contraintes, ainsi que les différentes stratégies que nous étudions. Développé en Objective Caml, il utilise des modules externes écrits en d'autres langages : Java, Prolog, C, etc. Ces modules ont été utilisés pour éviter de récrire des choses déjà existantes et testées.

Le prototype AuGuSTe fonctionne en trois étapes principales (voir Figure 6.1) : la construction de la structure combinatoire du graphe de contrôle, le tirage des éléments et des chemins, et la résolution de contraintes pour passer des chemins aux valeurs de test. Si la phase de résolution échoue, on retourne à l'étape de tirage. Dans la figure 6.1, les phrases en italique représentent les actions qui sont spécifiques aux critères "tous les enchaînements" et "toutes les instructions", les autres phrases sont valables pour tous les critères. Chaque étape et chaque passage d'une étape à l'autre est complètement automatisé.

Après avoir présenté les différents modules externes que nous utilisons, nous présentons en détail AuGuSTe.

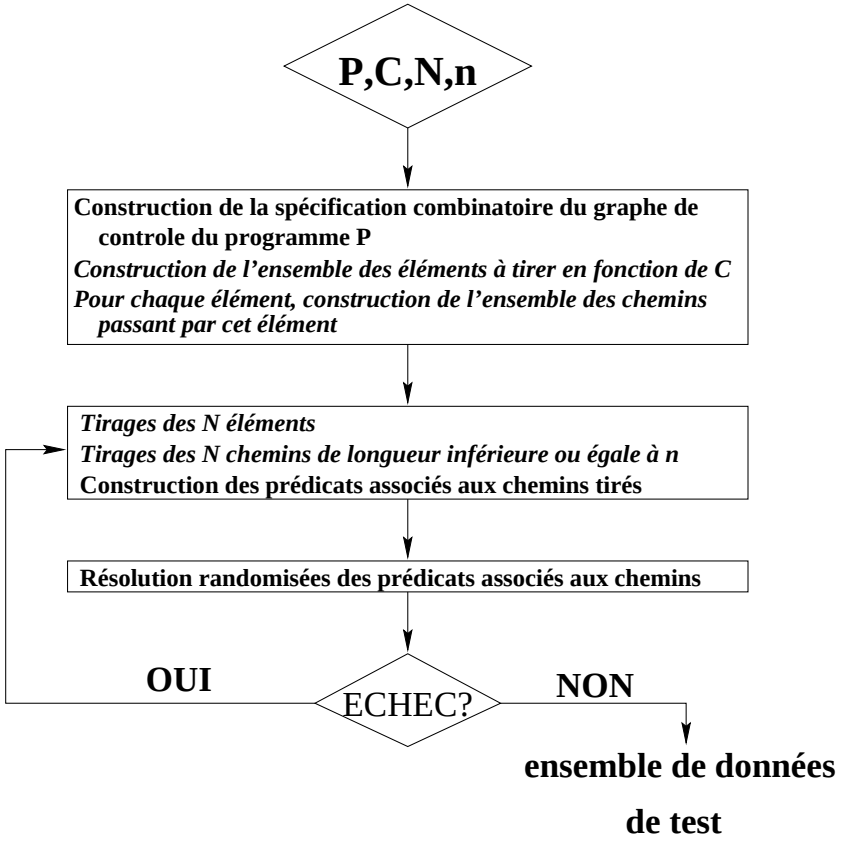


Fig. 6.1 : *Les différentes étapes du prototype AuGuSTe*

6.1 Les modules externes utilisés

Cette section est consacrée à l'introduction des outils et des modules que nous avons utilisés lors du développement de notre prototype.

L'environnement MuPAD est utilisé pour résoudre le problème linéaire permettant le calcul de la distribution sur les éléments de $E_C(D)$ (section 4.2.2) ainsi que pour la manipulation des structures combinatoires avec les librairies CS et MuPAD-Combinat.

Le package Java *dk.brics.automaton*, qui définit des automates à partir d'expressions régulières, est utilisé pour construire les différentes structures combinatoires dans la version 2 d'AuGuSTe.

Quant au module de résolution de GATeL, il est utilisé pour résoudre les prédicats des chemins.

6.1.1 Outils combinatoires

Pour manipuler les structures combinatoires, il existe deux librairies (CS et MuPAD-Combinat) qui ont été développées pour le logiciel MuPAD. Après avoir brièvement présenté MuPAD, nous décrivons ces deux librairies.

6.1.1.1 L'environnement MuPAD

MuPAD (*Multi Processing Algebra Data Tool* [109]) est un système de calcul algébrique qui manipule symboliquement plutôt que numériquement les expressions mathématiques. Parmi ses diverses fonctionnalités, il existe une fonction `linopt::maximize` qui permet de résoudre un système linéaire en maximisant la solution. Comme nous l'avons déjà vu en section 4.2.2, l'algorithme utilisé pour la résolution est l'algorithme du simplexe. Cette fonction prend trois arguments : le système linéaire, la variable à maximiser et des informations concernant la solution (par exemple, *NonNegative* pour imposer une solution positive). La solution obtenue est un triplet indiquant si la solution est optimale ou non, l'ensemble des variables avec leurs valeurs et la valeur de la variable maximisée. L'exemple 23 donne la syntaxe pour représenter un système linéaire à résoudre puis un exemple d'appel à `linopt::maximize` avec p comme variable à maximiser. La dernière commande de l'exemple permet de construire l'ensemble des variables non nulles du système : il s'agit de récupérer le deuxième argument de la solution et retirer l'ensemble des variables nulles.

Exemple 23 (MuPAD et la résolution linéaire)

```
>> monSysteme :={
  p<=e0+e1+e2+e3,
  p<=(1/5)*e0+(1/5)*e1+e2,
```

```
  p<=(3/5)*e0+(3/5)*e1+e3,
  1=e0+e1+e2+e3} ; ;

>> laSolution := linopt::maximize([monSysteme,p,NonNegative]) ; ;

   [OPTIMAL, {e0 = 0, e1 = 0, p = 1/2, e2 = 1/2, e3 = 1/2}, 1/2]


>> maSolution :=op(laSolution,2) minus e0=0,e1=0,e2=0,e3=0 ; ;

                        p = 1/2, e2 = 1/2, e3 = 1/2
```

MuPAD permet également d'écrire de véritables programmes (par exemple, les instructions *For* et *If* sont disponibles) avec gestion des entrées-sorties (lecture, écriture dans des fichiers).

6.1.1.2 Le package CS

La première librairie permettant la manipulation de structures combinatoires dans MuPAD est celle développée par Corteel, Denise, Dutour, Sarron et Zimmermann [33] : CS. Elle permet de définir des spécifications combinatoires, de compter et de générer aléatoirement des structures de taille fixée. CS produit aussi des fichiers en langage C contenant une spécification combinatoire et une fonction permettant de générer des objets de cette spécification.

Dans CS, les objets et les opérateurs se représentent de la façon suivante :
- l'objet vide se traduit par la constante `Epsilon`
- un objet de taille 1 est une suite de caractères non vide
- l'union disjointe + se traduit par l'opérateur n-aire `Union`
- le produit × se traduit par l'opérateur n-aire `Prod`
- les opérateurs *sequence*, *cycle* et *set* restent inchangés

Exemple 24 (Spécification dans CS de $S = \epsilon + a \times S \times b \times S$)

specExemple :={
  S=`Union`(`Epsilon`,*NT1*),
  NT1= `Prod`(*a*,*S*,*b*,*S*),
  a= `Atom`,
  b= `Atom`,
}

À partir d'une spécification combinatoire *spec* de source S, plusieurs opérations sont possibles dont :

`count([S,spec],size=k)` est une fonction de dénombrement qui permet de compter le nombre de structures combinatoires définies par la spécification combinatoire *spec* de source S et de taille k

`draw([S,spec],size=k)` produit aléatoirement et uniformément une structure combinatoire de taille k définie par la spécification combinatoire *spec* de source S

`compile(spec,target=C,file="fichier.c",main=source)` produit un fichier en langage C, *fichier.c*, contenant la spécification combinatoire et une fonction principale permettant de faire de la génération aléatoire de structures combinatoires de taille donnée. La commande MuPAD `system("cc -Dsys -o exe fichier.c -lm")` permet de compiler le fichier *fichier.c* en un exécutable `exe` directement utilisable en ligne de commande (c'est-à-dire sans utiliser MuPAD) dans le système *sys* (linux, sun, etc.).

Exemple 25 (Utilisation de `count`)
Soit specExemple la spécification combinatoire de $S = \epsilon + a \times S \times b \times S$.
Combien y a-t-il de structures combinatoires de longueur 6 ?

```
>> cs::count([S,specExemple],size=6) ;
```

```
5
```

Combien y a-t-il de structures combinatoires de taille de 1, 2, 3, 4 et 5 ?

```
>> cs::count([S,specExemple],size=i)$i=1..5 ;
```

```
0 1 0 2 0
```

Il existe, par exemple, 2 structures combinatoires de taille 4 qui sont :
`Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*),b,*`Epsilon`*)* *qui correspond à* $a \times a \times b \times b$ *et* `Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*))* *qui correspond à* $a \times b \times a \times b$.

Exemple 26 (Utilisation de `draw`)
Pour générer aléatoirement une structure combinatoire de taille 6, il faut utiliser la commande :

```
>> cs::draw([S,specExemple],size=6)
```

Qui nous donne :

`Prod`*(a,*`Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*),b,*`Epsilon`*),b,*`Epsilon`*)*

ou `Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*)),b,*`Epsilon`*)*

ou `Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*),b,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*))*

ou `Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*)))*

ou `Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*),b,*`Epsilon`*))*

Chaque structure combinatoire a une probabilité de $\frac{1}{5}$ *d'être engendrée.*

Pour engendrer aléatoirement des structures combinatoires de taille $\leq n$ d'une spécification combinatoire de source S, il suffit de modifier cette spécification en ajoutant une nouvelle source $S0$ et un atome "virtuel" `v` de la façon suivante :

S0= `Union`(S,`Prod`(v,S0))
S=`Union`(`Epsilon`,NT1)

puis de générer des structures combinatoires de taille n. En ignorant tous les atomes v, on obtient ainsi des structures combinatoires de taille $\leq n$.

Exemple 27 *Pour engendrer des structures combinatoires de longueur inférieure ou égale à 4 issues de la spécification specExemple, on modifie cette dernière :*
specExemple :={
S0= `Union`(S,`Prod`(v,S0)),
S=`Union`(`Epsilon`,NT1),
NT1= `Prod`*(a,S,b,S),*
a= `Atom`*,*
b= `Atom`*,*
v= `Atom`
}
puis on utilise la commande `draw` *:*
`cs::draw([S0,specExemple],size=4)`
Qui nous donne l'objet vide `Epsilon` *de taille 1 :*

`Prod`*(v,*`Prod`*(v,*`Prod`*(v,*`Epsilon`*)))*

ou l'objet $a \times b$ de taille 2 :

`Prod`*(v,*`Prod`*(v,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*)))*

ou l'objet $a \times b \times a \times b$ de taille 4 :

`Prod`*(a,*`Epsilon`*,b,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*))*

ou l'objet $a \times a \times b \times b$ de taille 4 :

`Prod`*(a,*`Prod`*(a,*`Epsilon`*,b,*`Epsilon`*),b,*`Epsilon`*)*

Chaque structure combinatoire a une probabilité de $\frac{1}{4}$ d'être engendrée.

Ce package est compatible avec les versions 1.3, 1.4 et 1.4.1 de MuPAD. C'est celui qui a été utilisé pour la première version d'AuGuSTe et donc pour les expériences.

Les versions 2.x ayant modifié considérablement le noyau du logiciel MuPAD, le package CS est devenu inutilisable. Nicolas Thiéry et ses collaborateurs ont mis à jour les fonctionnalités de ce package au sein de MuPAD-Combinat[113]. La version 2 d'AuGuSTe utilise cette nouvelle librairie.

6.1.1.3 Le package MuPAD-Combinat

La librairie MuPAD-Combinat contient de nombreuses fonctionnalités mais nous ne présentons ici que celles qui correspondent à nos besoins. Comme dans CS, cette librairie permet de définir une spécification combinatoire, de compter et générer des objets de taille fixe, et de créer des fichiers en langage C contenant une spécification combinatoire et une fonction de tirage.

Les algorithmes utilisés sont les mêmes que ceux de CS, seule la syntaxe change :

- Pour définir une structure combinatoire :

  $nom\_spec := \{Init = ...\}$
  est remplacé par
  $nom\_spec := \{Init = ...\}$
  $mon\_nom :=$
  $combinat :: decomposableObjects(nom\_spec, MainTerminal = Init)$

- Pour compter les objets de taille k :
  $cs :: count([Init, nom\_spec], size = k)$
  est remplacé par
  $mon\_nom :: count(k)$

- Pour tirer les objets de taille k :
  $cs :: draw([Init, nom\_spec], size = k)$
  est remplacé par
  $mon\_nom :: random(k)$

- Pour générer un fichier en langage C :
  $cs :: compile(nom\_spec, target = C, file = "fichier.c", main = Init)$
  est remplacé par
  $mon\_nom :: generateCode("fichier.c")$

6.1.2 Manipulation d'automates en Java

Cette section présente un package permettant la manipulation d'automates. Ces automates sont utilisés dans la deuxième version d'AuGuSTe pour construire l'ensemble des chemins qui passent par un élément ou un couple d'éléments $E_C(D)$. Chaque automate représente un ensemble de chemins (cf. section 6.3.2.2). À partir de ces ensembles, on peut construire la distribution des éléments de $E_C(D)$.

La librairie que nous utilisons, est le package Java *dk.brics.automaton* développé par Møller [81]. Ce package permet la construction d'automates à partir d'automates (par union, intersection, etc.) mais aussi d'expressions régulières. Il est composé de cinq classes : *Automaton.java* qui définit les automates, *RegExp.java*

qui définit les expressions régulières, *State.java* qui définit les états des automates, *StatePair.java* qui définit des couples d'états et *Transition.java* qui définit les transitions de l'automate. D'origine, les étiquettes des transitions ne supportaient que des caractères. Nous avons donc opéré des modifications : la première a consisté à modifier dans tous les fichiers les types *char* (qui correspondaient aux étiquettes) en type *int* et la seconde à définir d'autres méthodes `toString` pour traduire un automate en la spécification combinatoire correspondante.

Une fois les types des étiquettes modifiés, il a fallu revoir toutes les méthodes en particulier toutes celles liées aux méthodes `union`, `intersection`, `minimize` et `determinize`. Il nous a également fallu rajouter un opérateur de concaténation[1], noté @, entre expressions régulières afin de pouvoir distinguer[2] le symbole "33" du symbole "3" suivi du symbole "3".

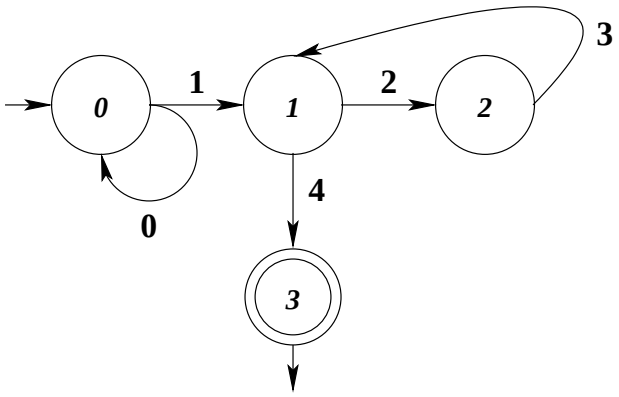


Fig. 6.2 : *Un automate reconnaissant le langage* $0^*1(2.3)^*4$

La méthode `toString` permet d'afficher l'automate de la figure 6.2 sous la forme suivante :

```
initial state : 0
state 0 [reject] :
      1 -> 2
      0 -> 0
state 1 [reject] :
      3 -> 3
state 2 [reject] :
      4 -> 3
      2 -> 1
state 3 [accept] :
```

[1]Le symbole "." est utilisé dans cette librairie pour représenter un symbole quelconque

[2]Traditionnellement, la concaténation de caractères se fait par juxtaposition *ab* au lieu de *a.b* avec le symbole . comme opérateur de concaténation.

où 0 est l'état initial, seul l'état 3 est acceptant et, par exemple, de l'état 2 (`state 2`) on peut passer à l'état 3 par la transition étiquetée 4 (`4 -> 3`).

Nous avons défini une nouvelle méthode `myString` permettant de représenter un automate sous forme la forme d'une spécification combinatoire (cf. section 6.3.3). Ainsi l'automate de la figure 6.2 peut être représenté sous la forme :

```
Init= S0,
S0=Union(Prod(a1,S2),Prod(a0,S0)),
S1=a3,
S2=Union(a4,Prod(a2,S1)),
a2= Atom,a4= Atom,a1= Atom,a3= Atom,a0= Atom
```

où $S0$ représente l'état initial 0 et $a0$ l'arc étiqueté 0 qui relie le sommet 0 à lui-même.

Un fichier d'interface entre les modules Objective Caml et Java a également été écrit. Sa fonction principale est décrite en section 6.3.2.2.

6.1.3 Module de résolution de GATeL

L'outil GATeL [87] que nous avons déjà présenté au chapitre 2 utilise une méthode de résolution de contraintes qui se différencie des méthodes classiques par ses heuristiques de choix (clauses, variables, valeur) efficaces et basées en partie sur la randomisation.

6.1.3.1 Résolution dans GATeL

La résolution dans GATeL a déjà été introduite page 26, nous ne rappelons ici que l'heuristique qui permet de choisir la variable à instancier. Ce choix s'effectue selon quatre critères qui sont dans l'ordre :

1. choisir une variable qui ne dépend fonctionnellement d'aucune autre variable. Ce critère donne la priorité aux valeurs initiales et correspond à une stratégie d'évaluation "par valeur"
2. choisir la variable avec le plus petit index[3] ceci dans le but de refléter l'évaluation séquentielle,
3. choisir la variable avec le plus petit domaine dans le but de minimiser le nombre de branchements dans l'arbre de résolution,
4. choisir la variable qui apparaît dans le maximum de contraintes.

[3]Chaque variable est numérotée selon son ordre d'apparition dans le programme. La variable d'index i apparaît plus tôt dans le programme que la variable d'index j, $j \geq i$

L'équation définissant une variable est introduite comme une contrainte uniquement lorsque cela s'avère nécessaire c'est-à-dire lorsque des contraintes sont en attente d'une valuation de cette variable. Cette propagation de contraintes "paresseuse" permet de minimiser le nombre moyen de contraintes, et donc la quantité de mémoire nécessaire.

De plus, la recherche d'une solution dans ce solveur de contraintes est limitée en nombre de *backtracks* (cf. section 1.2.3), ou retours arrière, autorisés et en temps. Ces limitations sont basées sur le principe suivant : si la résolution prend trop de temps et/ou qu'il y a trop de retours en arrière, alors il est possible que la branche explorée ne soit pas la bonne. De plus, cela permet d'avoir une résolution de prédicats dans un temps acceptable. Par contre, cela demande une gestion particulière des prédicats, dont on ne sait pas s'ils ont ou non une solution puisque la résolution s'est achevée prématurément.

6.1.3.2 Modifications apportées

La version du module de résolution de contraintes que nous avons utilisée présente des modifications par rapport au module original. La première a consisté à retirer la gestion des cycles que le langage LUSTRE nécessitait, et qui était dans notre cas inutile. Une deuxième modification porte sur la gestion des tableaux qui était absente dans GATeL et que nous avons dû ajouter. Dans notre cas, la taille des tableaux est bornée mais dans des travaux récents [88], qui utilisent également une adaptation du noyau de GATeL, il est possible de ne pas borner cette taille.

De plus, afin de différencier les échecs sûrs (pas de solution) des autres interruptions de la résolution, les sorties en cas d'échec ont également été différenciées. Ainsi la sortie `fail` indique que le prédicat n'a pas de solution et la sortie `again` indique que la résolution s'est arrêtée en cours de route. Un module d'interface spécifique à nos prédicats a donc été écrit.

Toutes ces modifications ont été réalisées par Bruno Marre.

6.1.3.3 Syntaxe utilisée

Un prédicat de chemin est représenté par une conjonction[4] de prédicats de la forme :

decl_entree(Nom,NumVar,Typ,NbreOcc) représente la déclaration de variable d'entrée *Nom*. Cette variable est identifiée[5] par le numéro *NumVar*, est de type *Typ* et est définie[6] *NbreOcc* fois dans le programme.

[4]Le symbole de conjonction dans ECL^{i}PS^e est la virgule.

[5]Chaque variable du programme qu'elle soit locale ou d'entrée, est associée à un numéro unique. Ce numéro est utilisé par le solveur de contraintes pour reconnaître les différentes variables

[6]L'occurrence 0 étant la première valeur de la variable.

decl_entree(Nom,NumVar,Typ,BorneInf,BorneSup,NbreOcc) représente la déclaration de la variable *Nom* d'entrée du programme. Cette variable est identifiée par le numéro *NumVar*, est de type tableau de *Typ* et dont les indices vont de *BorneInf* à *BorneSup* et est définie *NbreOcc* fois dans le programme.

decl(Nom,NumVar,Typ,NbreOcc) représente la déclaration de la variable locale *Nom* du programme. Cette variable est identifiée par le numéro *NumVar*, est de type *Typ* et est définie *NbreOcc* fois dans le programme.

decl(Nom,NumVar,Typ,BorneInf,BorneSup,NbreOcc) représente la déclaration de la variable *Nom* du programme. Ce variable est identifiée par le numéro *NumVar*, est de type tableau de *Typ* et dont les indices vont de *BorneInf* à *BorneSup* et est définie *NbreOcc* fois dans le programme.

def(var(Nom,Occ),Expr) représente l'affectation d'une variable. La variable *Nom* d'occurrence *Occ* prend la valeur de l'expression *Expr*.

def(var(Nom,Occ,ExprIndice),Expr) représente l'affectation d'une variable tableau. À l'indice *ExprIndice* du tableau *Nom* et d'occurrence *Occ*, on associe la valeur de l'expression *Expr*.

check(Expr) représente le fait de vérifier que l'expression *Expr* est vraie.

Les différentes déclarations doivent être considérées comme une phase d'initialisation du solveur : elles permettent de construire le domaine des valeurs de chaque variable et la structure qui représentera ses différentes occurrences. Les définitions permettent de compléter les structures qui associent à chaque couple (variable,occurrence) une valeur. Les *check* sont des conditions que doivent remplir les différentes valeurs choisies pour les différentes occurrences des variables. Le prédicat de l'exemple page 21, $(u_0{>}0)$ `and(`$v_0{>}u_0$`)and(`$t_0{=}u_0$`)and(`$u_1{=}v_0$`)and(`$v_1{=}t_0$`)and(`$u_2{=}u_1{-}v_1$`)and(`$u_2{>}0$`) and(`$\neg(v_1{>}u_2)$`)and(`$u_3{=}u_2{-}v_1$`)and(`$\neg(u_3{>}0)$`)`, sera représenté par les expressions suivantes :

$$
\begin{array}{lcl}
 & \Rightarrow & decl\_entree(u,0,Int,4),\\
 & \Rightarrow & decl\_entree(v,1,Int,2),\\
 & \Rightarrow & decl(t,2,Int,1),\\
u_0 > 0 & \Rightarrow & check(gt(var(u,0),val(0))),\\
v_0 > u_0 & \Rightarrow & check(gt(var(v,0),var(u,0))),\\
t_0 = u_0 & \Rightarrow & def(var(t,0),var(u,0)),\\
u_1 = v_0 & \Rightarrow & def(var(u,1),var(v,0)),\\
v_1 = t_0 & \Rightarrow & def(var(v,1),var(t,0)),\\
u_2 = u_1 - v_1 & \Rightarrow & def(var(u,2),minus(var(u,1),var(v,1))),\\
u_2 > 0 & \Rightarrow & check(gt(var(u,2),val(0))),\\
\neg(v_1 > u_2) & \Rightarrow & check(not(gt(var(v,1),var(u,2)))),\\
u_3 = u_2 - v_1 & \Rightarrow & def(var(u,3),minus(var(u,2),var(v,1))),\\
\neg(u_3 > 0) & \Rightarrow & check(not(gt(var(u,3),val(0))))
\end{array}
$$

6.2 Entrées et sorties du prototype

Les entrées du prototype sont le programme P à tester, le critère de couverture considéré C, le nombre de tests N et la longueur maximale n des chemins considérés. Les constructions de base sont :

- la composition séquentielle
- la construction conditionnelle *If...Then...Else* (où *Else* est optionnel)
- les boucles *While*
- les boucles *For* qui sont transformées en compositions séquentielles si le nombre d'itérations est connu ou en boucles *While* dans le cas contraire.

Les types de données utilisés sont les booléens, les entiers et les tableaux d'entiers ou de booléens. Actuellement, il n'y a pas de type enregistrement.

Exemple 28 (Le programme `rechercher_indice`)
Le programme `rechercher_indice` *prend en entrée un tableau* `t` *d'entiers trié en ordre croissant et un entier* `valeur`*. Il recherche la case du tableau dans laquelle est stockée cette* `valeur` *et renvoie le plus petit indice d'une telle case si il la trouve et renvoie* -1 *le cas contraire.*

```
int rechercher_indice(t : array 1..5 of int, valeur :int){
int i = 1 ;
int indice_recherche = -1 ;
while(i≤5){
  if(t[i]=valeur)
  then{
   indice_recherche := i ;
   i:=5 ;
 }
 else{
   if(t[i]>valeur)
   then {i := 5}
 }
  i :=i+1
}
return indice_recherche
}
```

Une analyse syntaxique et lexicale est effectuée afin de construire le graphe de contrôle du programme P passé en entrée. Le module qui réalise cette traduction est complètement indépendant du module principal de AuGuSTe, ceci permet de facilement changer le type de programme analysé.

En plus des constructions de base, nous avons ajouté une construction permettant l'insertion d'assertions. Ces assertions n'interviennent en aucune façon

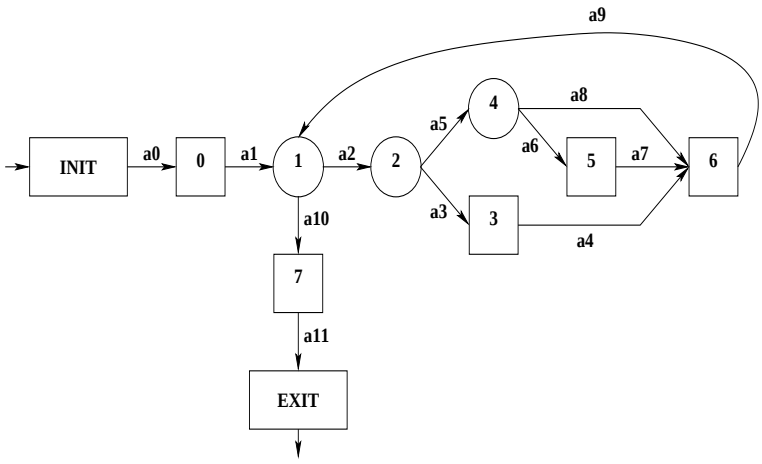


Fig. 6.3 : *Graphe de contrôle du programme* `recherche_indice`

dans la construction du graphe de contrôle. Elles ne sont utilisées que dans la phase de construction des prédicats de chemins pour préciser des contraintes liées à l'application considérée. Il existe deux types d'assertions :

- celles ajoutées en début de programme qui peuvent être utilisées, par exemple, pour préciser des propriétés sur des entrées (entier strictement positif).
- celles ajoutées comme argument supplémentaire aux boucles qui peuvent être utilisées pour préciser un intervalle d'itérations, par exemple, pour préciser que cette boucle ne peut-être itérée qu'exactement 18 ou 19 fois.

Le lecteur trouvera en annexe A la grammaire des programmes considérés.

La sortie principale d'AuGuSTe est un fichier contenant les N données de test mais il est également possible de récupérer l'ensemble des spécifications combinatoires construites ainsi que la distribution sur les éléments du critère utilisée.

6.3 Initialisation

La phase d'initialisation (cf. figure 6.4) se divise en trois étapes :

1. Analyse du programme, construction du graphe de contrôle et d'une table d'association T
2. En fonction du critère, construction de la distribution et des ensembles de chemins considérés
3. Traduction des ensembles de chemins en les spécifications combinatoires correspondantes

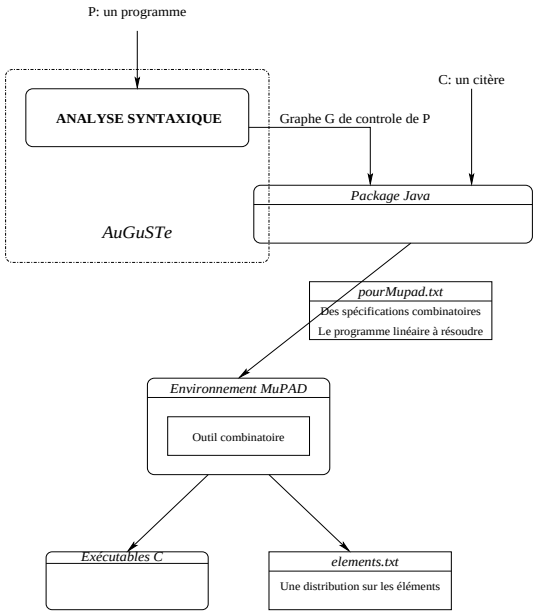


Fig. 6.4 : *Phase d'initialisation du prototype AuGuSTe (version 2)*

6.3.1 Analyse du programme

Deux modules (*grammaire.mly*, *lexer.mll*) sont chargés d'analyser syntaxiquement le programme, puis de le traduire en une structure de données `programme` ayant les champs suivants :

- `entrees` est une liste de couples (nom,type) qui représente les entrées du programme
- `variables` est une liste de couples (nom,type) qui représente les variables locales du programme
- `corps` est une structure de données représentant les instructions qui composent le programme

Exemple 29 (Structure de données `rechercher_indice`)
Les champs suivants, en Objective Caml, correspondent à ceux de la structure de données du programme de l'exemple 28.

`entrees`= *[(t,array(1,5,int)) ;(valeur,int)]*

`variables`= *[(i,int) ;(indice_recherche,int)]*

`corps`= *[Affect(Var(i),Int(1)) ;Affect(Var(indice_recherche),Int(-1)) ;*
While(Leq(Var(i),Int(5)),
[If(Eq(Tab(t,Var(i)),Var(valeur)),
[Affect(Var(indice_recherche),Var(i)) ;Affect(Var(i),Int(5))],
[If(Gt(Tab(t,Var(i)),Var(valeur)),

[Affect(Var(i),Int(5))],[])];
Affect(Var(i),Add(Var(i),Int(1)))])];
Return(Var(indice_recherche))]

Ensuite, un module (*graphe.ml*) parcourt la liste d'instructions du programme (`programme.corps`) en construisant le graphe de contrôle G et une table d'association T.

La structure de données utilisée pour représenter le graphe G se présente dans AuGuSTe sous forme de listes. Ce choix est entièrement lié à la possibilité dans Objective Caml de faire facilement de la reconnaissance de motifs (*pattern-matching*) et aux premiers algorithmes que nous avons utilisés. Pour un autre langage, et peut-être même pour les prochaines versions d'AuGuSTe, d'autres structures seraient sûrement utilisées. Un graphe G est donc une liste dont les éléments sont :

- $Bloc(i)$ qui correspond à un bloc d'instructions indivisibles et étiqueté i ;
- $While(i, l)$ qui correspond à une boucle que l'on a étiquetée i. Le branchement de la boucle est représenté par une liste l d'éléments ;
- $If(i, lt, le)$ qui correspond à une instruction conditionnelle *IfThen* ou *IfThenElse* et étiquetée i. Les instructions de la branche *Then* sont représentés par la liste lt et ceux de la branche *Else* par la liste le. Dans le cas d'un *IfThen*, la liste le est vide.

Les listes l, lt et le contiennent comme la liste de G des $Bloc(i)$, des $While(i, l')$ et des $If(i, lt', le')$.

Nous avons choisi de représenter un chemin sous forme d'une suite d'arcs. Par conséquent, la table T associe chaque instruction (pré-traduite en prédicat) à un arc. Pour les conditions des instructions conditionnelles (*While*, *IfThen*, *IfThenElse*), la condition est associée à un arc et sa négation (sortie de boucle, passage dans le *Else*) à un deuxième.

Exemple 30 (G et T)
La structure de données suivante représente le graphe de contrôle du programme `rechercher_indice` *:*

[Init;Bloc(0);While(1,[If(2,[Bloc(3)],[If(4,[Bloc(5)],[])]);Bloc(6)]);Bloc(7);Exit]

où Bloc représente une séquence d'instructions non conditionnelles.

La table d'association T correspondante est présentée en figure 6.5. L'opérateur `def` *représente une affectation qui modifiera l'occurrence de la variable affectée lors de la construction du prédicat final (cf. sections 1.2.2 et 6.5). Les ? représentent les valeurs des occurrences considérées, ils seront instanciés lors de la construction du prédicat.*

Arc	Prédicat
0	true
1	def(var(i, ?),val(1)),def(var(indice_recherche, ?),val(-1))
2	leq(var(i, ?),val(5))
3	eq(var(t,var(i, ?), ?),var(valeur, ?))
4	def(var(indice_recherche, ?),var(i, ?)),def(var(i, ?),val(5))
5	not(eq(var(t,var(i, ?), ?),var(valeur, ?)))
6	gt(var(t,var(i, ?), ?),var(valeur, ?))
7	def(var(i, ?),val(5))
8	not(gt(var(t,var(i, ?), ?),var(valeur, ?)))
9	def(var(i, ?),add(var(i, ?),val(1)))
10	not(leq(var(i, ?),val(5)))
11	true

Fig. 6.5 : *Table d'association T de l'exemple 30*

6.3.2 Construction de la distribution

Si le critère choisi est "tous les chemins", la spécification combinatoire correspondant au graphe de contrôle est calculée à l'aide de l'algorithme présenté en section 3.2. Puis le fichier C associé est généré et compilé en un fichier exécutable appelé `chemins`.

Sinon, selon la méthode utilisée pour construire la distribution sur les éléments, cette partie diffère d'une version à l'autre du prototype. Nous présentons d'abord la version utilisée (section 6.3.2.1) pour les expériences puis la dernière version développée (section 6.3.2.2). Cette dernière version est celle qui sera utilisée dans la suite de nos travaux.

6.3.2.1 Construire un ensemble d'éléments

L'heuristique de la première version consistait à construire un ensemble S à partir des feuilles d'un arbre des dominants étendu (voir section 5.2.2.1). Cet ensemble S était complété par la racine de l'arbre si les sommets de S ne permettaient pas d'atteindre tous les chemins du graphe de contrôle.

Dans le prototype AuGuSTe, cette heuristique a été implémentée par un algorithme qui parcourt le graphe de contrôle en récupérant les sommets ou les arcs recherchés. La structure de données utilisée pour représenter le graphe de contrôle G permet de récupérer facilement ces éléments. En effet, par construction, ces éléments sont associés à des listes vides ou contenant un $Bloc(..)$. Les sommets sont obtenus immédiatement : ce sont ceux qui sont dans les listes à un élément. Les arcs sont soit ceux qui sortent de ces sommets[7], soit ceux qui

[7]Les sommets considérés pouvant avoir plusieurs arcs entrants, c'est-à-dire qui arrivent à ce

matérialisent l'enchaînement correspondant à l'absence de *Bloc* (i.e. []).

Parallèlement, un marquage des chemins couverts est maintenu à jour, afin de vérifier que tous les chemins sont couverts. S'il existe un chemin non marqué, l'algorithme ajoute le premier sommet (critère "toutes les instructions") ou le premier arc (critère "tous les enchaînements") du graphe de contrôle.

Exemple 31 (Construction de l'ensemble S)
Pour le programme `rechercher_indice`*, les sommets aux feuilles sont ceux qui correspondent aux Blocs 3 et 5. Quant aux arcs, il faut prendre un représentant de chaque classe aux feuilles :* {3,4}, {6,7} *et* {8}*. L'arc 4 (resp. 7) est un arc sortant du sommet correspondant au* $Bloc(3)$ *(resp.* $Bloc(5)$*). L'arc 8 correspond à l'enchaînement qui relie le sommet correspondant au* $If(4,..,..)$ *au sommet correspondant au* $Bloc(6)$*, "en passant par la liste vide []".*

Dans ces deux cas, l'ensemble S doit être complété par le sommet correspondant au $Bloc(0)$ *par l'arc 0 (qui relie le sommet Init au sommet du* $Bloc(0)$*).*

Une spécification combinatoire est alors construite pour pouvoir utiliser le tirage uniforme :

$$ensembleS := \{\texttt{Union}(e_1, e_2, ..., e_{|S|})\} \text{ où } \forall i, e_i \in S$$

Ensuite, pour chaque élément e de S, un sous-graphe G_e de G représentant l'ensemble des chemins de G passant par l'élément e est construit. Puis, l'algorithme décrit en section 6.3.3 permet de construire une spécification combinatoire correspondant à chaque G_e. La spécification combinatoire `ensembleS` ainsi que toutes les spécifications combinatoires associées aux graphes G_e sont traduites dans des fichiers C puis compilés. Au final, $|S| + 1$ exécutables ont été produits :

- `ensembleS` permet le tirage parmi les éléments de S
- pour chaque élément e_i de S, `avec_ei` permet le tirage parmi les chemins passant par e_i.

6.3.2.2 Construire une distribution sur les éléments

Le calcul de la distribution sur les éléments, décrit au chapitre 4, nécessite plusieurs constructions intermédiaires. Ces constructions intermédiaires permettent de calculer le nombre de chemins passant par un élément e_i, les α_i, ou par un couple d'éléments (e_i, e_j), les α_{ij}.

L'étape décrite dans cette section se déroule de la manière suivante :

1. tous les ensembles de chemins à considérer sont construits à l'aide d'automates puis traduits en une spécification combinatoire correspondante.

sommet, nous avons choisi de prendre l'arc sortant qui est unique pour tous les sommets de S.

2. à l'aide des outils combinatoires décrits en section 6.1.1, chaque α_{ij} est calculé
3. le programme linéaire est alors produit et résolu avec la méthode du simplexe, donnant ainsi une distribution sur les éléments e_i
4. en fonction de la couverture des chemins obtenue avec cette distribution, on modifie ou non le programme linéaire précédent

Construction des automates Pour commencer, un module (*regexp.ml*) traduit le graphe G en l'expression régulière correspondante. Cette expression régulière a pour alphabet l'ensemble des arcs du graphe. Chaque mot du langage décrit par cette expression régulière est un chemin du graphe. Pour notre exemple, l'expression régulière obtenue est :

$$a0.a1.(a2.(a3.a4|a5.(a6.a7|a8)).a9) * .a10.a11$$

Cette expression régulière et l'ensemble des éléments à couvrir, noté $E_C(D)$ au chapitre 4, sont utilisés pour construire des automates finis à l'aide des modules de la librairie Java (cf. section 6.1.2).

L'expression régulière, passée en paramètre du constructeur de la classe `Automate`, permet de construire l'automate déterministe et minimal A_G correspondant au graphe de contrôle.

Remarque 2 *Par la suite, on considérera tous les automates comme étant déterministes et minimaux. Ils ont forcément ces qualités car chaque transition porte une étiquette différente.*

Ensuite, pour chaque élément e de $E_C(D)$, on construit un automate correspondant à l'ensemble des chemins passant par e, ce qui revient à calculer l'automate correspondant à l'expression régulière $\mathcal{A}^* e \mathcal{A}^*$ où $\mathcal{A}$ est l'alphabet. Puis, par intersection avec l'automate A_G, on construit l'automate A_e de l'ensemble des chemins de G passant par e.

Remarque 3 *Dans le cas du critère "tous les enchaînements", l'expression régulière $\mathcal{A}^* e \mathcal{A}^*$ est immédiate puisque l'alphabet $\mathcal{A}$ est l'ensemble des arcs du graphe de contrôle. Par contre dans le cas du critère "toutes les instructions", l'arc e utilisé correspond au* **seul** *arc ayant comme origine (ou destination) le sommet considéré.*

Ensuite, pour chaque couple d'éléments (e_i,e_j), on construit l'automate A_{e_i,e_j} résultat de l'intersection de A_{e_i} avec A_{e_j}. L'automate A_{e_i,e_j} reconnaît tous les mots qui sont des chemins du graphe de contrôle passant au moins une fois par e_i et au moins une fois par e_j.

Une fois tous ces automates construits (tous ces ensembles de chemins), il nous faut calculer les α_{ij} (nombre de chemins de G passant par e_i et e_j) pour construire le programme linéaire. Dans AuGuSTe, nous avons choisi d'utiliser la fonction `count` de CS (resp. MuPAD-Combinat) pour calculer les α_{ij} et le solveur linéaire de MuPAD pour résoudre le programme linéaire.

Remarque 4 *Les automates A_{e_i,e_j} ne servent qu'à calculer les α_{ij}. Il y existe d'autres méthodes pour calculer ces α_{ij} (avec les séries génératrices par exemple) mais pour des raisons d'homogénéité avec ce qui est fait par ailleurs dans AuGuSTe nous avons privilégié cette approche.*

Pour construire la spécification combinatoire correspondant à un automate, nous utilisons la méthode `myString` que nous avons définie en section 6.1.2.

Calcul des α_{ij} Un fichier temporaire *pourMupad.txt* destiné à MuPAD, contient toutes ces spécifications combinatoires. Pour chacune de ces spécifications, un appel à une fonction de dénombrement avec comme paramètre la longueur maximale n est rajouté : le résultat étant stocké dans une variable c_e_e'. La variable c_e_e' représente le nombre de chemins de longueur inférieure ou égale à n du graphe G qui passent à la fois par e et e'. Le programme linéaire est alors ajouté en entrée de MuPAD.

Le fichier se termine par un programme MuPAD qui permet de récupérer les éléments aux probabilités non nulles et de les stocker avec leur probabilité dans le fichier *elements.txt*. De plus, pour chacun de ces éléments, la spécification combinatoire correspondant à l'ensemble des chemins de G passant par cet élément est traduite en un programme C correspondant et compilé.

Couverture des chemins Pour vérifier que tous les chemins du graphe G sont couverts, il suffit de récupérer tous les éléments de probabilités non nulles et de faire l'union de leurs automates, ce qui nous donne un automate A. Nous testons ensuite l'égalité[8] des deux A_G et A, qui sont minimaux et déterministes. S'ils sont égaux alors tous les chemins sont couverts.

Avec la librairie d'automates que nous utilisons dans AuGuSTe, il n'est pas possible de faire une différence d'automates ou encore de calculer un automate complémentaire. Si ces fonctions étaient à disposition, il suffirait de faire l'automate A_G moins A ou encore l'intersection de A_G avec l'automate complémentaire $\bar{A}$ de A. Dans le cas où l'automate obtenu reconnaît le langage vide, cela implique que tous les chemins[9] de G sont couverts (avec une probabilité non nulle).

[8] Il est également possible ici de remplacer A_G par l'union de tous les automates A_e, avec $e \in E_C(D)$

[9] Pour ne considérer que les chemins couvrant les éléments de $E_C(D)$, remplacer A_G par un automate obtenu en faisant l'union de tous les automates A_e, avec $e \in E_C(D)$

Comme nous l'avons vu en section 4.2.3, il existe plusieurs solutions dans les cas où au moins un chemin n'est pas couvert. Dans AuGuSTe, nous avons implémenté comme première solution, celle qui consiste à construire un nouveau programme linéaire en ajoutant la contrainte[10] que l'arc 0 doit être de probabilité non nulle, de le résoudre et d'obtenir ainsi un nouveau fichier *elements.txt* assurant la couverture de tous les chemins. Dan la prochaine version d'AuGuSTe, l'utilisateur pourra choisir entre les différentes solutions proposées en section 4.2.3.

6.3.3 D'un graphe de contrôle à une spécification combinatoire

Cette section explique, comment pour chaque sous-graphe de contrôle (ensemble de chemins), la spécification combinatoire est construite.

La transformation d'un graphe de contrôle en une spécification combinatoire s'effectue de manière récursive et automatique à partir de la structure de données représentant le graphe G (et ses sous-graphes) sous forme de listes (cf. section 6.3.1). Cette transformation est basée sur celle décrite en section 3.2, nous avons choisi ici comme objets atomiques les arcs du graphe de contrôle plutôt que ses sommets.

Nous présentons ici les quatre cas de figure qui correspondent à la composition séquentielle, à la construction *Boucle*, à la construction conditionnelle *If...Then...Else* et à la construction conditionnelle *If...Then*.

Cas de la composition séquentielle La première construction (Fig. 6.6) est celle décrite par un sommet I suivi d'un graphe quelconque S. La structure combinatoire est $I = \times(i, S)$ qui se traduit en CS par :

$$I = \texttt{Prod}(i, S)$$

Une optimisation permet d'écrire directement $I = i$ si le bloc I est le sommet particulier du graphe de contrôle c'est-à-dire soit le sommet est suivi de $Exit$ soit ce sommet correspond à un $Bloc(n)$ tel que l'on ait $[Bloc(n)]$ dans le graphe de contrôle. Le sommet $Exit$ n'a pas besoin d'être représenté.

Exemple 32

$[Init; Bloc(0); While(1, [If(2, [Bloc(3)], [If(4, Bloc(5), [])]); Bloc(6)]); Bloc(7); Exit]$
se traduit en $Init = Prod(a0, S)$
où S *est la transformation en spécification combinatoire du sous-graphe*
$[Bloc(0); While(1, [If(2, [Bloc(3)], [If(4, [Bloc(5)], [])]); Bloc(6)]); Bloc(7); Exit]$

[10] La contrainte actuelle fixe la probabilité π_0 de tirer l'arc 0 à $\frac{1}{\alpha\_0\_0}$

$[Bloc(3)]$ *se traduit en* $I3 = a4$.

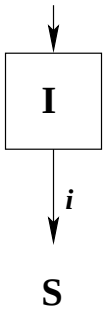


Fig. 6.6 : *Cas d'un bloc d'instruction*

Cas de la boucle La deuxième construction est celle qui décrit un cycle dans le graphe qui est une représentation classique d'une boucle dans le programme (Fig. 6.7). Sur le graphe de la figure 6.7, S et C représente des sous-graphes (pour l'instant inconnus) correspondant respectivement à la suite du programme et au corps de la boucle. À ce niveau de la construction de la spécification combinatoire, seuls les arcs s et b sont connus.

La structure combinatoire correspondante à une boucle est

$$B = +(\times(s, S), \times(b, C, B))$$

qui se traduit en CS par :

$$B = \texttt{Union}(\texttt{Prod}(s, S), \texttt{Prod}(b, C, B))$$

Une optimisation permet décrire directement $B = \texttt{Union}(s, \texttt{Prod}(b, C, B))$ si le programme se termine par cette boucle, on a alors $[While(., .., ..); Exit]$ ou bien si la boucle est en "fin de liste" dans le graphe de contrôle c'est-à-dire que l'on a $[While(., .., ..)]$.

Exemple 33
$[While(1, [If(2, [Bloc(3)], [If(4, [Bloc(5)], [])]); Bloc(6)]); Bloc(7); Exit]$
se traduit en $B1 = Union(Prod(a10, S), Prod(a2, C, B1))$
où S *est la représentation combinatoire de* $[Bloc(7); Exit]$
et C *celle de* $[If(2, [Bloc(3)], [If(4, [Bloc(5)], [])]); Bloc(6)]$

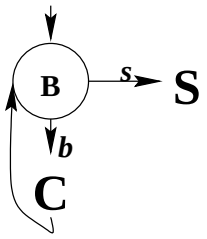


Fig. 6.7 : *Cas d'une boucle*

Cas d'une condition incomplète La troisième construction est celle correspondant à une instruction *IfThen* (Fig. 6.8). La structure combinatoire est $\times(+(\times(t,T),e),S)$ (ou $+(\times(e,S),\times(t,T,S))$) qui se traduit en CS par :

$$C = \texttt{Prod}(\texttt{Union}(\texttt{Prod}(t,T),e),S)$$

Une optimisation permet décrire directement $C = \texttt{Union}(\texttt{Prod}(t,T),e)$ si le programme se termine sur cette construction conditionnelle, on a $[If(.,..,[]);Exit]$ ou bien si cette instruction conditionnelle est en "fin de liste" dans le graphe de contrôle c'est-à-dire que l'on a $[If(.,..,[])]$.

Exemple 34
$[If(4,[Bloc(5)],[])]$ *se traduit en* $C4 = Union(Prod(a6,T),a8)$
où T *est la représentation combinatoire de* $[Bloc(5)]$.

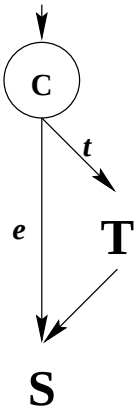


Fig. 6.8 : *Cas d'une condition incomplète*

Cas d'une condition complète La dernière construction (Fig. 6.9) est celle correspondant à une instruction *IfThenElse*. La structure combinatoire correspondante est $\times(+(\times(t,T),\times(e,E)),S)$ (ou $+(\times(t,T,S),\times(e,E,S))$) qui se traduit en CS par :

$$C = \texttt{Prod}(\texttt{Union}(\texttt{Prod}(t,T),\texttt{Prod}(e,E)),S)$$

Une optimisation permet décrire directement $C = \texttt{Union}(\texttt{Prod}(t,T),\texttt{Prod}(e,E))$ si le programme se termine sur cette construction conditionnelle, on a $[If(.,..,..);Exit]$ ou bien si cette instruction conditionnelle est en "fin de liste" dans le graphe de contrôle c'est-à-dire que l'on a $[If(.,..,..)]$.

Exemple 35
$[If(2,[Bloc(3)],[If(4,[Bloc(5)],[])]);Bloc(6)]$ *se traduit en*
$C2 = Prod(Union(Prod(a3,T),(a5,E)),S)$
où T *est la représentation combinatoire de* $[Bloc(3)]$, E *celle de* $[If(4,[Bloc(5)],[])]$ *et* S *celle de* $[Bloc(6)]$.

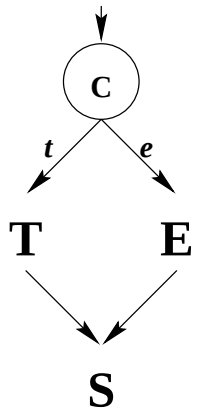


Fig. 6.9 : *Cas d'une condition complète*

Exemple 36 (Spécification combinatoire)
La spécification combinatoire suivante représente l'ensemble des chemins du graphe de contrôle du programme `rechercher_indice` *:*

```
chemins :={
Init= Prod(a0, I0),
I0= Prod(a1,B1),
B1=Union(Prod(a10,I7),Prod(a2,C2,B1)),
C2= Union(Prod(a3, I3), Prod(a5, C4)),
I3= a4,
```

```
C4= Prod(Union(Prod(a6, I5), a8), I6),
I5= a7,
I6= a9,
I7= a11,
a0= Atom, a1= Atom, a2= Atom, a3= Atom, a4= Atom, a5= Atom,
a6= Atom, a7= Atom, a8= Atom, a9= Atom, a10= Atom, a11= Atom } ; ;
```

6.4 Les tirages

Nous expliquons maintenant comment sont effectués les différents tirages (chemins et éléments) dans le prototype AuGuSTe.

Si le critère choisi est "tous les chemins", on utilisera le fichier exécutable `chemins` (cf. section 6.3.2). Ainsi pour tirer N chemins de longueur inférieure ou égale à n, le prototype fait un appel à cet exécutable :

`chemins` n N

Deux modules (*grammaireCS.mly*, *lexerCS.mll*) analysent ce qui est retourné par l'exécutable et le transforment en une liste de N chemins. Un chemin est une liste de k arcs (avec $k \leq n$) car les arcs "virtuels" v qui ne sont d'aucune utilité pour construire le prédicat associé à ce chemin sont supprimés.

Pour les critères "tous les enchaînements" et "toutes les instructions", il y a deux types différents de tirages : celui qui porte sur les éléments et celui qui porte sur les chemins. Nous avons vu dans les sections 6.3.2 et 5.2 que la distribution sur les éléments de $E_C(D)$ était différente dans les deux versions d'AuGuSTe.

Dans la première version, il s'agit d'un tirage uniforme dans un sous-ensemble S. Ce tirage est effectué de la même façon que celui des chemins. Ainsi pour tirer N éléments, il suffit d'appeler l'exécutable `ensembleS` avec comme arguments N et 1. Deux modules (*grammaireElement.mly*, *lexerElement.mll*) analysent la sortie de cet exécutable et la transforment en une liste de N éléments.

Ensuite, cette liste de N éléments est utilisée pour construire[11] une liste de couples (élément,occurrences) qui à chaque élément associe le nombre de fois qu'il a été tiré, et donc le nombre de chemins passant par cet élément qu'il faudra tirer. Puis, pour chaque couple (e_i, k), on appelle l'exécutable `avec_ei` avec les arguments i k. Des modules (*grammaireCS.mly*, *lexerCS.mll*) sont utilisés à chaque tirage pour récupérer la sortie de l'exécutable et construire la liste de k chemins. En faisant la concaténation de toutes ces listes, on obtient au final une liste de N chemins.

[11] Cette construction n'est pas obligatoire mais elle permet d'optimiser le tirage des chemins. En effet, au lieu de tirer k fois un chemin, on va pouvoir tirer une fois k chemins.

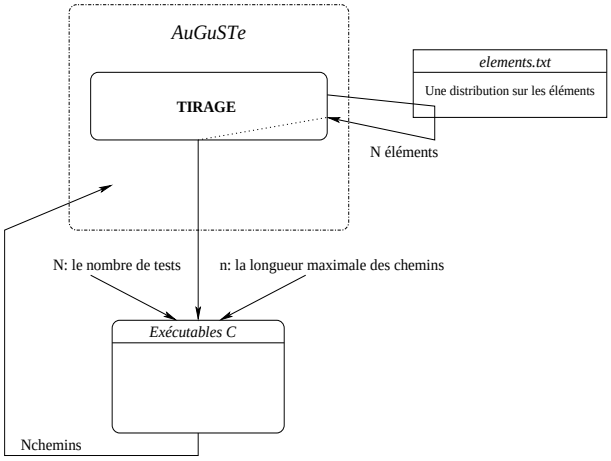


Fig. 6.10 : *Phase de tirage du prototype AuGuSTe (version 2)*

Dans la deuxième version (cf. figure 6.10), par contre, il faut tenir compte de la distribution construite à l'étape 1. Cette distribution a été récupérée en faisant une analyse du fichier *elements.txt* (cf. section 6.3.2.2) à l'aide des modules *grammaireElementProba.mly* et *lexerElementProba.mll*, ce qui permet de récupérer une liste de couples (élément,probabilité). Ensuite, en utilisant la fonction `Random.float(1.0)` disponible en Objective Caml, on peut tirer un nombre flottant entre 0 et 1, puis l'associer à l'un des éléments de la liste. En effectuant N tirages de nombre flottant, on peut ainsi obtenir une liste de N éléments (de probabilité non nulle) de $E_C(D)$. À partir d'ici, le traitement de ces N éléments est le même que dans la version 1 à savoir :

- construction d'une liste de couples (éléments, occurrences) ;
- pour chaque couple (e_i, k), appelle de la commande "`avec_ei n k`" ;
- concaténation des listes de chemins pour construire la liste finale de N chemins.

À la fin de cette étape, quelque soit le critère considéré, nous avons une liste de N chemins de longueur inférieure ou égale à n.

6.5 Construction et résolution des prédicats de chemin

À partir des chemins tirés, nous devons récupérer les entrées qui permettront l'exécution de ces chemins. Pour cela, il nous faut construire et résoudre pour chaque chemin le prédicat qui lui est associé (cf. figure 6.11).

Pour construire les prédicats, nous utilisons la table d'association T (cf. section 6.3) et l'algorithme classique présenté dans la section 1.2.2. Un parcours

récursif du chemin va être effectué. Une table d'association T' associant à chaque variable du programme l'occurrence courante est maintenue à jour. Prenons par exemple le chemin $a0a1a10a11$ et la table d'association T de la section 6.3.1. Au départ $T' = \{(t, 0), (valeur, 0)\}$, les variables d'entrée ayant une valeur de départ, leur première occurrence est 0. Quant aux variables locales, comme elles n'ont pas encore été initialisées, elles n'apparaissent pas encore dans T'. Notons $pred(a0a1a10q11)$, le prédicat associé à la suite d'arcs $a0a1a10a11$ que nous allons construire récursivement dans la syntaxe décrite en section 6.1.3.3.

Première étape : $pred(a0a1a10a11) := true, pred(a1a10a11)$ car dans T, l'arc 0 est associé à $true$;

Deuxième étape :

$pred(a0a1a10a11) :=$

$$def(var(i, 0), val(1)), def(var(indice\_recherche, 0), val(-1)), pred(a10a11)$$

À la fin de cette étape, la table T' est complétée en rajoutant les couples $(i, 0)$ et $(indice\_recherche, 0)$ car les variables i et $indice\_recherche$ viennent d'être initialisées.

Troisième étape :

$pred(a0a1a10a11) :=$

$$def(var(i, 0), val(1)), def(var(indice\_recherche, 0), val(-1)),$$
$$not(leq(var(i, 0), val(5))),$$
$$pred(a11)$$

car l'occurrence actuelle de la variable i est 0.

Quatrième étape :

$pred(a0a1a10a11) :=$

$$def(var(i, 0), val(1)), def(var(indice\_recherche, 0), val(-1)),$$
$$not(leq(var(i, 0), val(5))), true$$

Dernière étape : On rajoute au prédicat construit un préambule qui correspond à la déclaration des variables du programme. Les occurrences sont déterminées en fonction de T'. Dans notre exemple, le préambule serait de la forme :

$$decl\_entree(t, 0, Int, 1, 5, 1),$$
$$decl\_entree(valeur, 1, Int, 1),$$
$$decl(i, 2, Int, 1),$$
$$decl(indice\_recherche, 3, Int, 1)$$

Le prédicat associé au chemin $a0a1a10a11$ est :

$$decl\_entree(t, 0, Int, 1, 5, 1),$$
$$decl\_entree(valeur, 1, Int, 1),$$
$$decl(i, 2, Int, 1),$$
$$decl(indice\_recherche, 3, Int, 1),$$
$$def(var(i, 0), val(1)), def(var(indice\_recherche, 0), val(-1)),$$
$$not(leq(var(i, 0), val(5)))$$

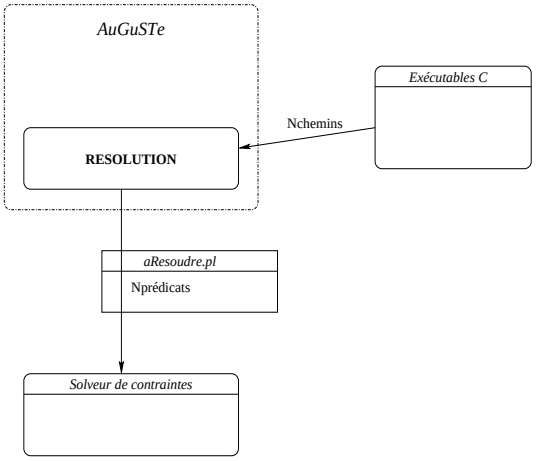


Fig. 6.11 : *Phase de resolution du prototype AuGuSTe (version 2)*

Ce procédé est répété sur tous les chemins en prenant soin de réinitialiser la table d'association T' à chaque fois. Le préambule est le même modulo le nombre d'occurrences de chaque variable.

Une fois tous les prédicats construits, on construit un fichier *aresoudre.pl* contenant ces prédicats ainsi que les appels à la fonction de résolution correspondant à chaque prédicat. Ce fichier est chargé dans l'environnement ECL^{i}PS^e. Les sorties sont récupérées à l'aide de deux modules (*grammairePL.mly* et *lexerPL.mll*) puis analysées.

6.6 Analyse du résultat de la résolution

Une fois la résolution des prédicats effectuée, une analyse des résultats (cf. figure 6.12) permet de trier les prédicats en trois catégories :

- ceux qui ont une solution : c'est le cas le plus simple. La solution est la donnée de test qui permet d'exécuter le chemin associé.
- ceux qui n'ont pas de solution : le chemin associé est prouvé infaisable
- ceux dont on ne sait pas s'ils ont ou non une solution

Dans le premier cas, nous avons donc les tests. Les deux derniers cas demandent une gestion particulière.

6.6.1 Gestion des prédicats insatisfiables

Si un prédicat est insatisfiable, c'est que le chemin associé est infaisable. Pour gérer les chemins infaisables, il existe plusieurs stratégies possibles (voir section 4.4).

Dans une première version, le prototype AuGuSTe enregistrait en mémoire ces chemins infaisables puis retournait à l'étape 1. Lors de l'étape 2, si un chemin tiré était répertorié comme étant un chemin infaisable connu, le prototype AuGuSTe le rejetait aussitôt et en retirait un nouveau. En présence de trop de chemins infaisables, le prototype est confronté à un problème de mémoire.

La deuxième version, utilisée pour les expériences, n'utilise aucune stratégie de mémorisation pour la gestion des chemins infaisables et se contente de retourner à l'étape 1.

6.6.2 Gestion des prédicats indécis

Dans le cas où la résolution s'est arrêtée avant la fin, il existe plusieurs manières de gérer le problème.

Une première consiste à retenter une résolution de ce prédicat en utilisant le fait que la résolution est randomisée et en espérant que la probabilité de retomber sur les mêmes valeurs soit faible.

Une autre consiste à abandonner et retourner à l'étape 1.

Dans AuGuSTe, nous avons opté pour la seconde solution. Ce choix arbitraire permet de gérer ces chemins comme les chemins infaisables.

Notons que dans nos expériences, avec un nombre de *backtrack* limité à 10 et un temps de 10 secondes, nous avons constaté que très peu de prédicats, moins de 1%, se retrouvaient dans ce cas indécis.

6.7 Conclusion

AuGuSTe est un exemple de prototype que l'on peut développer à partir de notre approche. Tout prototype issu d'une application de notre approche à une méthode de test se décompose en 4 étapes (Fig. 6.1) :

1. – Construction d'une spécification combinatoire correspondant à l'ensemble des chemins de la description D passée en paramètre.
   – *Construction de la distribution permettant de couvrir avec la probabilité maximale tous les éléments du critère C passé en paramètre.*

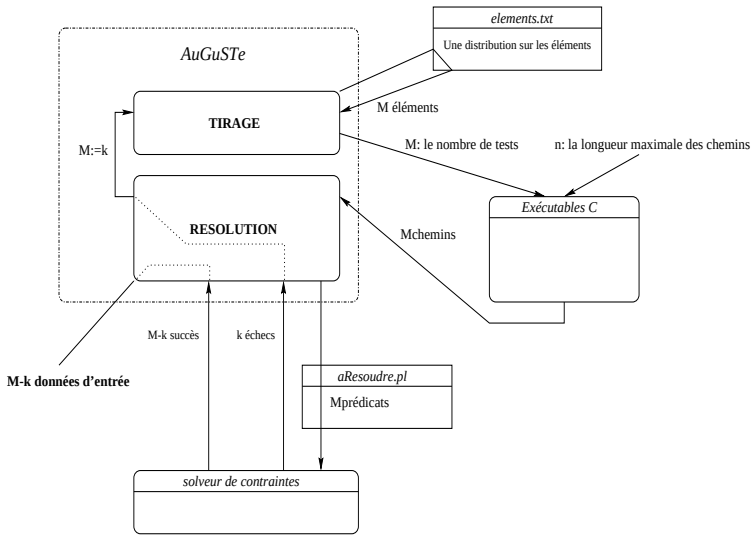


Fig. 6.12 : *Phase d'analyse du résultat de la résolution du prototype AuGuSTe (version 2)*

- *Pour chaque élément du critère, construction des spécifications combinatoires correspondant à l'ensemble des chemins de la description D qui passent par cet élément.*

2. – *Tirage des éléments*
   - Tirage des chemins
   - Construction des prédicats associé aux chemins tirés
3. Résolution des prédicats associés aux chemins tirés
4. En fonction des résultats de la résolution, retour éventuel à l'étape 2

Les entrées d'AuGuSTe sont le programme à tester, le critère de couverture, la longueur maximale des chemins à considérer et le nombre de tests N à effectuer. Sa sortie est un fichier contenant les N données de test. Il est toutefois possible de récupérer l'ensemble des spécifications combinatoires construites ainsi que la distribution sur les éléments du critère utilisée.

Lors d'un stage de licence, Pierre-Loic Garoche a développé pour AuGuSTe une interface graphique permettant entre autre la visualisation graphique du graphe de contrôle du programme testé. L'intérêt de cette visualisation est de permettre éventuellement l'intervention du testeur en choisissant par exemple les sommets (resp. arcs) qu'il voudra ou non tester plus spécifiquement. Comme nous le verrons au chapitre suivant, cela peut être très intéressant.

La prochaine version d'AuGuSTe devrait être plus flexible sur le choix des différentes stratégies notamment dans le cas où la distribution calculée par le programme linéaire ne permet pas de couvrir tous les chemins ou revient à faire du

tirage uniforme parmi tous les chemins du graphe. De plus, on pourrait également permettre de remplacer le nombre de test N par la qualité de test q souhaitée puis calculer le nombre de tests N.

Chapitre 7

Résultats expérimentaux

Ce chapitre présente les expériences et les résultats que nous avons obtenus et qui nous ont permis d'évaluer notre approche. Les expériences que nous avons réalisées avec notre prototype, et qui sont présentées dans ce chapitre ont plusieurs objectifs :

1. évaluer le pouvoir de détection des fautes de notre méthode. Pour cela, nous avons comparé, pour un ensemble d'expériences, le nombre de fautes détectées par notre méthode par rapport à celui de la méthode du LAAS.
2. évaluer la stabilité du pouvoir de détection des fautes de notre méthode : comme toute méthode basée sur un tirage aléatoire, il faut s'assurer que les résultats obtenus ne sont pas le fruit d'un tirage heureux mais qu'ils restent semblables d'une expérience à l'autre.
3. estimer si un passage à l'échelle est possible : beaucoup de méthodes sont efficaces sur de petits programmes mais deviennent inopérantes dès que les programmes deviennent de taille réelle.

Pour ces expériences, nous avons utilisé la première version du prototype AuGuSTe, c'est-à-dire celle qui utilise la distribution des éléments basée sur la notion de dominance (section 5.2.2). Une section est cependant consacrée aux premières expériences réalisées sur le programme le plus complexe avec la distribution calculée de manière à optimiser la qualité de test (section 4.2.2).

Rappelons que notre prototype AuGuSTe est basé sur le tirage uniforme de chemins et la résolution randomisée de prédicats (cf. section 1.2.3.4), alors que la méthode du LAAS est basée sur la construction d'une distribution sur le domaine des entrées. Dans notre cas, nous ne pouvons pas établir la distribution du domaine des entrées qui correspond aux tests obtenus. En effet, le solveur de contraintes induit une distribution sur le domaine des entrées non explicitable car elle dépend :

– des stratégies de résolution et des heuristiques de simplifications du solveur,

– de la manière dont les prédicats sont écrits dans le programme.

La distribution du domaine des entrées ne peut pas servir de base à une comparaison avec la méthode de la section 1.4. Par ailleurs, la distribution implicite obtenue est forcément différente de celle du LAAS puisqu'elle est basée sur des chemins de longueur bornée (à l'avance) et donc ne couvre pas toujours tout le domaine d'entrée. C'est le prix à payer pour automatiser en évitant les constructions empiriques que la méthode du LAAS entraîne dans les cas complexes.

Après avoir introduit la technique d'évaluation d'une méthode de test à l'aide du test par mutation, nous présentons les expériences (programmes, mutants, longueurs des test) effectuées ainsi que les résultat obtenus par la méthode de test statistique structurel du LAAS. Puis nous présentons nos résultats ainsi que les difficultés que nous avons rencontrées au cours de ces expériences. Nous terminons ce chapitre par un bilan de cette campagne d'évaluation.

7.1 Évaluation d'une méthode de test à l'aide de mutants

Utilisé à l'origine comme critère de sélection [41], le test de mutation est également utilisé pour évaluer l'efficacité des méthodes de test dynamique [58].

C'est une méthode dite d'injection de fautes qui consiste à créer des *mutants* du programme à tester.

Définition 15 (Mutant d'après [40])
Un programme M est un mutant d'un programme P si M est construit à partir de P en introduisant **une et une seule** *modification élémentaire (mutation) dans le code de P. Cette modification est issue d'une liste de fautes de référence définies pour le langage de programmation du programme P.*

Chacun des mutants est exécuté sur un jeu de test J. Deux cas sont alors possibles :

– il existe au moins une entrée du jeu de test J telle que le mutant retourne un résultat différent que le programme original : dans ce cas, on dit que le mutant est **tué** par le jeu de test J.

– tous les résultats retournés par le mutant sont les mêmes que ceux retournés par le programme original : dans ce cas, on dit que le jeu de test J n'a pas réussi à tuer le mutant.

Exemple 37 (Quelques mutants pour le programme `pgcd`)

```
int pgcd(u,v:int)
t: int;
while (u>0){
 if (v>u)
  then{
   t:=u;
   u:=v;
   v:=t
  }
  u:=u-v;
}
return(v);
```

```
int mutant1(u,v:int)
t: int;
while (u≥0){
 if (v>u)
  then{
   t:=u;
   u:=v;
   v:=t
  }
  u:=u-v;
}
return(v);
```

```
int mutant2(u,v:int)
t: int;
while (u>0){
 if (v>u)
  then{
   t:=v;
   u:=v;
   v:=t
 }
  u:=u-v;
}
return(v);
```

```
int mutant3(u,v:int)
t: int;
while (u>0){
 if (v>u)
  then{
   t:=u;
   u:=v;
   v:=t
  }
 u:=u+v;
}
return(v);
```

Le point faible du test de mutation est le nombre de mutants. En effet, s'il faut construire un mutant pour chaque opération, chaque variable, chaque type, etc., on aboutit rapidement à un nombre impraticable de mutants. Les mutants doivent donc être créés avec discernement. Il existe différents outils comme Mothra [39] et SESAME [34] qui permettent de générer automatiquement un ensemble de mutants pour un programme donné. Dans notre étude, nous avons pu récupérer les programmes et tous les mutants utilisés pour l'évaluation du test statistique structurel [117].

Un autre problème du test de mutation consiste à déterminer si un mutant est équivalent à son programme d'origine ou non. En effet, dans le cas où aucun test n'a pu tuer un mutant, il faut déterminer s'il est possible de trouver un test qui pourrait le tuer. S'il en existe un, le mutant est un mutant **vivant** sinon c'est un mutant équivalent. L'équivalence de deux programmes est un problème indécidable, c'est pourquoi généralement la détermination de l'équivalence est

faite à la main. Les archives du LAAS ne comportant pas la liste des mutants équivalents, il nous a fallu les retrouver manuellement.

Exemple 38 (Un programme et un mutant équivalents)

```
int exemple(v:int)
t: int;
if (v>0)
 then
  t:=v;
 else
  t:=0;
return(t);
```

```
int mutant(v:int)
t: int;
if (v≥0)
 then
  t:=v;
 else
  t:=0;
return(t);
```

Un jeu de test, et par extension la méthode qui l'a généré, est évalué en mesurant le pourcentage de mutants tués, appelé aussi **score de mutation**.

Définition 16 (Score de mutation [117, 38]) *Un score de mutation pour un jeu de test T et un programme P est la proportion de mutants* **non équivalents** *de P qui sont tués par le jeu T.*

C'est un nombre compris entre 0 et 1 tel que plus il est proche de 0 moins le jeu de test s'est montré efficace pour révéler des fautes.

7.2 Programmes et mutants utilisés

Les expériences décrites dans [112] ont porté sur le test unitaire de quatre fonctions issues d'un logiciel industriel dont le descriptif est le suivant :

- FCT1 et FCT2 permettent l'acquisition de données
- FCT3 permet le filtrage de ces données
- FCT4 permet la conversion de ces données

Le tableau 7.1 récapitule le profil de chacune de ces fonctions c'est-à-dire son nombre de lignes de code, son nombre de chemins, le nombre de blocs et d'arcs qui composent son graphe de contrôle ainsi que le nombre de points de choix (*While*, *IfThen*, *IfThenElse*).

Pour les trois premières fonctions FCT1, FCT2 et FCT3, le critère structurel le plus fort a été appliqué, soit le critère "tous les chemins". En effet, ces fonctions ont un nombre fini de chemins. La fonction FCT4 possède quant à elle une boucle, c'est pourquoi le critère "tous les enchaînements" a été préféré.

Les fonctions simples FCT1 et FCT2 ont subi une seule série de tests. Pour le programme FCT3 qui dépend fortement de l'environnement (présence de variables

	♯lignes	♯chemins	♯blocs	♯arcs	♯choix
FCT1	30	17	14	24	5
FCT2	43	9	12	20	4
FCT3	135	33	19	41	12
FCT4	77	∞	19	41	12

Tab. 7.1 : *Les quatre fonctions testées*

non initialisées par exemple), 5 séries de tests ont été effectuées comme pour le programme FCT4 qui contient une boucle.

Le nombre de tests nécessaires pour chacune de ces fonctions a été calculé afin d'obtenir une qualité de test égale à 0.9999. Pour FCT1, FCT2 et FCT3, le nombre théorique de tests à effectuer doit être supérieur ou égal à 152, 79 et 300. Les expériences du LAAS ont porté sur une comparaison de différentes méthodes de test : le test déterministe, le test aléatoire et le test statistique structurel. Afin de pouvoir comparer ces méthodes sur des jeux de longueurs égales, le nombre de tests réalisés pour chaque programme a donc été déterminé en prenant le maximum des nombres de tests nécessaires pour chacune des trois méthodes, soit : 170 pour FCT1, 80 pour FCT2 et 405 pour FCT3.

Pour FCT4, le nombre de tests N a été obtenu en utilisant la méthode analytique (cf. section 1.4) avec le critère "toutes les utilisations" soit 850. La fonction FCT4 ayant la propriété suivante : la qualité de test fournie par un jeu de longueur N est identique quelque soit le critère, le nombre de tests à effectuer pour couvrir le critère "tous les enchaînements" a donc été de 850.

	critère	♯série	♯tests N par série
FCT1	tous les chemins	1	170
FCT2	tous les chemins	1	80
FCT3	tous les chemins	5	405
FCT4	tous les enchaînements	5	850

Tab. 7.2 : *Nombres de tests réalisés*

Les expériences ont été réalisées sur 2914 mutants fournis par l'outil SESAME [34]. Pour chaque fonction, le nombre de mutants diffère selon la complexité de son code et de sa taille : ainsi il y a 279 mutants pour FCT1, 563 pour FCT2, 1467 pour FCT3 et 605 pour FCT4. Les mutants sont de 3 types :

- **type C** : mutation de constantes
- **type O** : mutation d'opérateurs
- **type S** : mutation de symboles

	♯mutants		
	♯typeC	♯typeO	♯typeS
FCT1	122	57	100
FCT2	282	129	152
FCT3	618	432	417
FCT4	290	171	144

Tab. 7.3 : *Répartition des mutants pour chaque fonction*

Parmi ces mutants, on a classiquement des mutants équivalents. Ces mutants équivalents se divisent en trois catégories [117] :

- équivalence fonctionnelle : la mutation n'affecte ni la logique ni le comportement dynamique du programme (exemple 38) ;
- masquage systématique d'erreur : la mutation est une faute qui ne génère pas de défaillances observables (la redondance du code permet d'arriver à la même sortie par plusieurs chemins différents). Prenons l'exemple suivant, un programme contient le code :

```
if (m>0 and m<64)
then
 if (t[m]<>0)
 then return true
 else return false
else return false
```

  Supposons que d'après la spécification $t[0]$ soit toujours nul, alors la mutation de $m > 0$ en $m > -1$ n'entraîne pas d'effet observable et le mutant non équivalent ne peut être détecté.
- équivalence dépendante de l'environnement : le comportement dynamique reste inchangé alors que la logique du programme est affectée. Les propriétés de l'environnement sont telles que de telles fautes peuvent être masquées. Par exemple en C, les booléens sont représentés par des entiers : 0 pour faux et tout entier non nul pour vrai. Si l'initialisation `valeur_bool=1` est remplacée dans un mutant par la comparaison `valeur_bool==1`, alors la variable sera initialisée par une valeur arbitraire de la mémoire qui a très peu de chance d'être 0. Le mutant ne sera pas détecté en général. Le fait de découvrir ou non cette mutation peut varier d'une exécution à l'autre sur la même machine puisque l'initialisation dépend de l'état mémoire courant. Par contre, si le compilateur force l'initialisation de toute variable à 0 ou mieux, contraint toute variable à être initialisée explicitement (comme en Java), alors ce mutant a toutes les chances d'être tué.
  L'ensemble des mutants dont l'équivalence dépend de l'environnement peut

être différent d'un environnement (système d'exploitation, compilateur ou même version de celui-ci) à l'autre ou même d'une suite d'exécution à l'autre.

Le tableau 7.4 donne la répartition des mutants équivalents des deux premières catégories pour les quatre programmes considérés. Au total, il y avait également 48 mutants dépendant de l'environnement pour l'ensemble de ces quatre programmes.

	Équivalence fonctionnelle	Masquage systématique d'erreur
FCT1	14	0
FCT2	6	0
FCT3	14	7
FCT4	6	3

Tab. 7.4 : *Répartition des mutants par type d'équivalence*

L'efficacité d'un jeu de test pour une fonction FCTn s'évalue à l'aide de son score de mutation.

7.3 Les résultats du LAAS

	♯mutants non équivalents	score de mutation	♯mutants non tués
FCT1	265	1	0
FCT2	548	1	0
FCT3	1416	1	0
FCT4	587	min=0.9898 moy=0.9901 max= 0.9915	6

Tab. 7.5 : *Résultats du test statistique structurel*

Le tableau 7.5 récapitule les résultats obtenus par le test statistique structurel proposé par le LAAS pour ces quatre programmes. Pour les trois premiers programmes FCT1, FCT2 et FCT3, la méthode a tué tous les mutants non équivalents. Pour le dernier programme FCT4, même si les résultats ne sont pas parfaits, le score moyen est très bon. Les 6 fautes qui n'ont pas été mises en évidence par cette méthode dépendent de l'activation des bornes d'un tableau qui, d'après Waeselynck [117], pourraient être éliminées par un test déterministe[1].

[1]Le test des valeurs aux limites fait partie de la stratégie de test du LAAS

C'est à partir de ces résultats que nous allons comparer notre nouvelle méthode par rapport au test statistique structurel.

7.4 Résultats de notre approche

Nous avons repris les expériences du LAAS et nous présentons dans cette section les résultats que nous avons obtenus pour les trois premières fonctions.

La longueur n des chemins choisie pour chacune de ces fonctions correspond à la longueur du plus long chemin (élémentaire)[2] du graphe de contrôle de ces fonctions, soit 16, 14 et 16, respectivement pour FCT1, FCT2 et FCT3.

Dans ce chapitre, nous présentons nos premières expériences mais le lecteur pourra trouver en annexe B d'autres expériences réalisées et dont l'analyse est actuellement en cours.

7.4.1 Les programmes FCT1 et FCT2

Le tableau 7.6 récapitule les résultats que nous avons obtenus pour les programmes FCT1 et FCT2.

	score de mutation	p_{min} calculé	qualité de test
FCT1	1	$\frac{1}{17} = 0.0588$	0.9999
FCT2	1	$\frac{1}{9} = 0.1111$	0.9999

Tab. 7.6 : *Résultats expérimentaux pour FCT1 et FCT2*

FCT1 et FCT2, qui sont des programmes simples, n'ont posé aucun problème. Comme pour le LAAS, tous les mutants non équivalents ont été tués.

7.4.2 Le programme FCT3

Le tableau 7.7 présente les résultats obtenus par FCT3 pour cinq jeux de test obtenus à partir d'un p_{min} de $\frac{1}{33}$ et d'une qualité de test de 0.9999. Pour FCT3, la présence de variables globales partagées et non initialisées rend l'exécution des tests très dépendante de l'environnement et en particulier de l'ordre dans lequel on exécute les tests. C'est pourquoi tous les jeux ne se sont pas comportés de la même façon avec tous les mutants non équivalents : 3 jeux sur 5 ont obtenu un score de mutation parfait soit 1 et les 2 autres jeux ont laissé respectivement 1 et 7 mutants vivants. Ce qui nous donne pour le score de mutation une moyenne de 0.9989 avec un écart type de 0.0019.

[2]Tout n plus grand est possible mais l'ensemble de chemins reste identique

FCT3	score de mutation
série 1	0.9951
série 2	0.9993
série 3	1
série 4	1
série 5	1

Tab. 7.7 : *Résultats expérimentaux pour FCT3*

Quelle que soit la méthode de test utilisée, déterminer toutes les erreurs dans un programme fortement dépendant de l'environnement n'est pas une chose facile. En effet, son comportement n'est pas prévisible d'un jeu à l'autre et pire, on ne peut pas assurer que son comportement sera stable si on l'exécute plusieurs fois sur un même jeu de test. La stabilité de notre approche s'avère très satisfaisante. Remarquons que ce type de problème peut être détecté avant le test par une analyse systématique de flots de données.

7.5 Le cas particulier de FCT4

Le programme FCT4 est le plus intéressant des quatre programmes considérés. En effet, c'est le seul à avoir une boucle et donc une infinité de chemins. Il a soulevé de nombreux problèmes qui nous ont amenés à ajuster notre première approche et faire plusieurs séries d'expériences.

Après avoir décrit les particularités de ce programme, nous présenterons les différentes expériences réalisées et les résultats obtenus.

7.5.1 Particularités de FCT4

La structure générale du programme FCT4 se divise en deux parties :

– une instruction conditionnelle qui permet d'initialiser une variable

$$NB\_VOIES \text{ à } 18 \text{ ou } 19$$

– l'instruction principale qui est une boucle de condition

$$NUM\_VOIES < NB\_VOIES$$

où la variable NUM_VOIES est initialisée à 0 puis incrémentée de 1 à chaque itération.

Tout chemin ne faisant pas exactement 18 ou 19 itérations de boucle est donc infaisable. Le corps de la boucle possédait deux particularités :

- il existait du code mort que nous avons retiré
- il contenait une instruction conditionnelle `IfThenElse` dont la condition était $SE\_VOIE\_EN\_TEST = TRUE$ où la variable $SE\_VOIE\_EN\_TEST$ n'évolue pas avec les itérations.

Cette instruction conditionnelle est également une source de chemins infaisables.

La figure 7.1 représente le graphe de contrôle de ce programme après avoir retiré l'ensemble du code mort qui le composait. La présence de la boucle oblige à déterminer plus précisément la taille des chemins : en effet, il faut trouver un bon compromis entre une taille qui ne nous permettrait pas d'atteindre tous les chemins élémentaires et une taille qui couvrirait "inutilement" beaucoup trop de chemins.

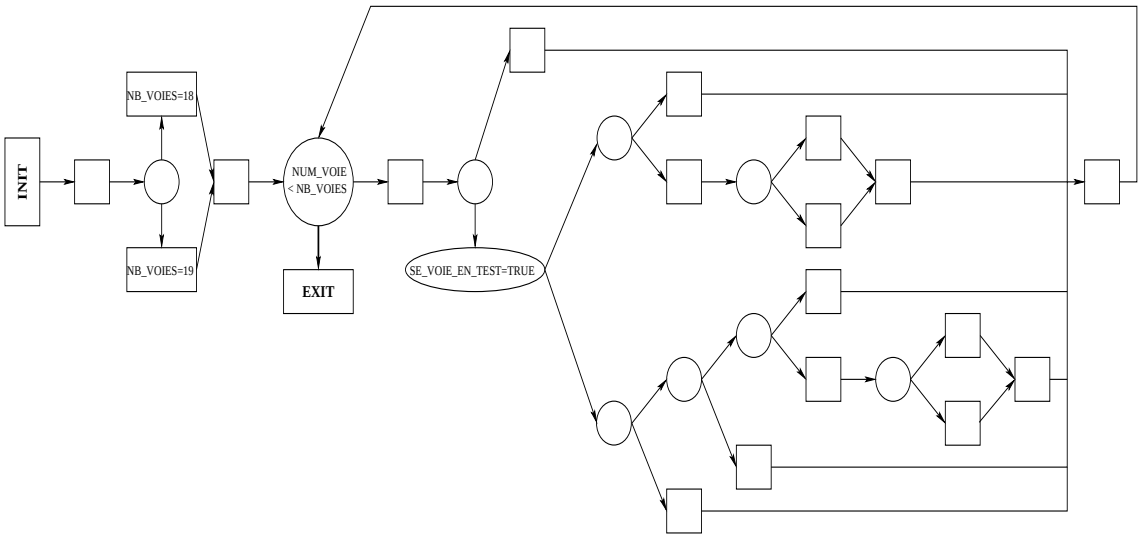


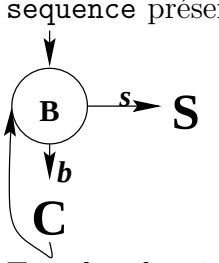
Fig. 7.1 : *Graphe de contrôle annoté du programme FCT4 (sans code mort)*

Dans notre cas particulier, le nombre possible d'itérations de la boucle étant limité à exactement 18 ou 19 passages, nous avons pu déterminer au plus juste la longueur maximale des chemins, il s'agit de la somme des 3 longueurs suivantes :

- $L_1 = 5$: longueur du plus long chemin élémentaire du début du programme au début de la boucle
- $L_2 = 12 \times 19$: longueur du plus long chemin élémentaire du corps de la boucle, que l'on a multiplié par le nombre maximum de fois que peut être itérée la boucle soit 19
- $L_3 = 1$: longueur du plus long chemin élémentaire qui va de la boucle à la fin du programme

Ce qui nous donne 234.

En ce qui concerne la spécification combinatoire associée à ce programme, deux choix sont possibles : soit ses propriétés sont incluses dans la spécification (notamment celle qui concerne le nombre d'itérations), soit elles sont ignorées. Dans un premier temps, nous avons voulu réaliser des expériences dans lesquelles ces propriétés n'étaient pas représentées. Cela nous donnait alors près de 10^{30} chemins de longueur inférieure ou égale à 234, dont une très grande majorité de chemins infaisables : en particulier tous ceux qui ne passaient pas exactement 18 ou 19 fois dans la boucle. Afin de diminuer le nombre de chemins infaisables et ainsi obtenir un jeu de test en un temps raisonnable, nous avons dû intégrer la propriété de la boucle dans la spécification combinatoire en utilisant l'opérateur `sequence` présenté en section 3.1.1.2 et en fixant la cardinalité à 18 et 19 :



$$B = s \times S + b \times C$$
$$\Rightarrow$$
$$B = s \times S + sequence(b \times C, card = 18) + sequence(b \times C, card = 19)$$

Tous les chemins ne passant pas 18 ou 19 fois dans la boucle sont alors écartés et il ne reste plus "que" 10^{20} chemins de longueur inférieure ou égale à 234. La proportion de chemins infaisables reste quand même très importante, elle est de l'ordre de 99,98% : lors d'une expérience consistant à tirer 850 chemins qui couvrent le critère "tous les enchaînements", parmi les 4 232 437 chemins tirés, 4 231 587 étaient infaisables.

7.5.2 Les expériences sur FCT4 et les dominances

Cette section présente les expériences réalisées à l'aide de la première version du prototype, c'est-à-dire celle dont la distribution sur les éléments est définie à partir de la notion de dominance. Trois types d'expériences ont été réalisées pour ce programme :

Première expérience : Les tests ont été obtenus à partir du programme FCT4 (sans le code mort) et de la structure combinatoire reflétant exactement tous les chemins du graphe de contrôle, avec la contrainte du nombre d'itérations de la boucle ;

Deuxième expérience : Les tests ont été obtenus à partir du programme FCT4 (sans le code mort) et d'une structure combinatoire modifiée ;

Troisième expérience : Les tests ont été effectués sur une version modifiée de FCT4 c'est-à-dire ne contenant pas de code mort et dont la structure combinatoire est celle utilisée pour l'expérience précédente, ainsi que sur les mutants correspondants.

Nous présentons ici ces trois expériences, leurs motivations et leurs résultats.

7.5.2.1 Première expérience

Cette expérience est celle pour laquelle nous sommes le moins intervenus. Elle consiste à fournir à AuGuSTe le programme et les différents paramètres, et à attendre les tests.

Les résultats d'un premier jeu de test ont été satisfaisants : 64 mutants non équivalents ont été laissés vivants. Ce qui nous donne un score de mutation de 0.9726. 49 de ces mutants correspondaient à des mutations que la spécification du programme rend équivalents (exemple : mutation dans le code mort).

En ce qui concerne les 15 mutants restants, une analyse des enchaînements couverts montre que certains enchaînements n'ont été couverts par aucun chemin. Ceci s'explique essentiellement par la présence de nombreux chemins infaisables, leur répartition, et leur gestion par le prototype.

L'ensemble des chemins infaisables reste très important à cause de la conditionnelle *IfThenElse* avec $SE\_VOIE\_EN\_TEST = TRUE$.

Dans une première version, le prototype AuGuSTe gérait les chemins dont la résolution avait échoué en refaisant pour chacun de ces chemins un tirage parmi les éléments du critère, puis pour chaque élément sélectionné, un tirage parmi les chemins qui passaient par l'élément. Le défaut de cette méthode réside dans le fait qu'un élément qui a trop peu de chemins faisables qui passent par lui risque de ne jamais être couvert. Pour corriger ce problème et accélérer le processus[3] de génération des tests, nous avons essayé deux stratégies.

La première consiste à modifier la gestion des chemins en échec. Au lieu de retirer parmi les éléments du critère, on choisit pour chaque chemin en échec de refaire un tirage dans l'ensemble où il a été choisi. Ou encore, cela revient à :

1. Tirer N éléments dans S
2. Pour chaque élément e_i de S, tirer dans l'ensemble des chemins qui passent par e_i jusqu'à atteindre un chemin faisable

Cette modification n'est possible que sous l'hypothèse qu'il n'y ait **pas de code mort** : il existe au moins un chemin faisable qui passe par chaque élément. Théoriquement, les résultats devraient être meilleurs, malheureusement cette stratégie rend la génération d'un jeu de tests beaucoup plus longue[4] surtout si le nombre de chemins faisables passant par certains éléments est très faible.

La seconde stratégie s'inspire les techniques d'optimisation de code et est à l'origine de la deuxième série d'expériences.

[3] Il nous a fallu une semaine pour obtenir un jeu de 850 tests.

[4] les premiers essais inachevés avaient atteints les 15 jours de recherche sans atteindre la moitié des 850 tests désirés

7.5.2.2 Deuxième expérience

Sans modifier le programme source, nous avons utilisé les techniques d'optimisation pour modifier la spécification combinatoire. Cela n'est possible que parce que le programme FCT4 s'y prête bien : en effet, la variable d'entrée `SE_VOIE_EN_TEST` n'étant jamais modifiée, il est alors possible de faire sortir ce test de la boucle. C'est une variante d'une technique classique d'optimisation de programme qui consiste à sortir des boucles les affectations de variables dont l'expression est invariante [9]. Une telle modification implique une transformation du graphe de contrôle (voir figure 7.2). En revanche, le programme qui sera effectivement testé n'a pas à être modifié.

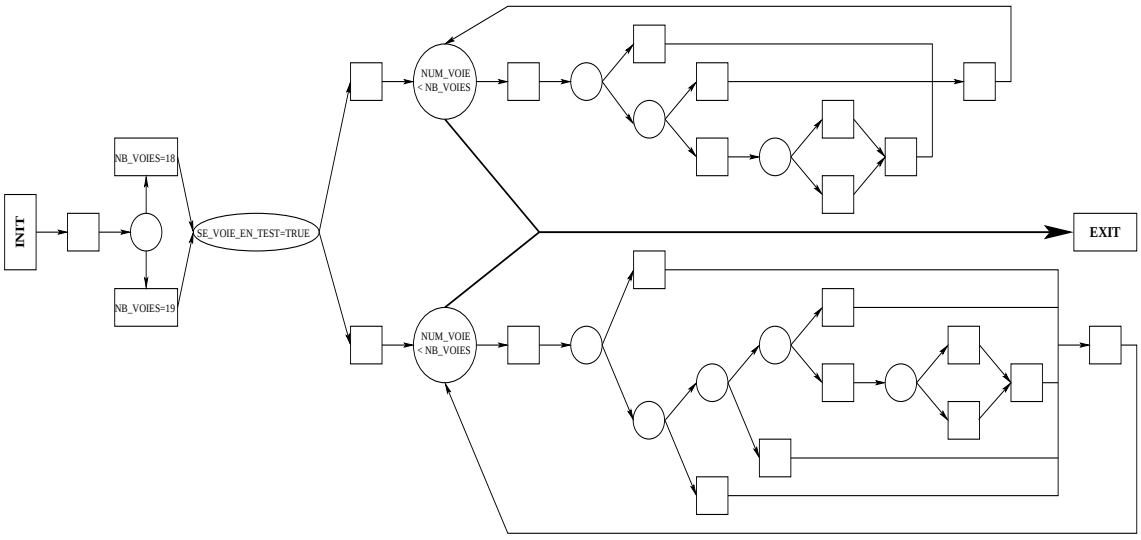


Fig. 7.2 : *Graphe de contrôle modifié, annoté et du programme FCT4 (sans code mort)*

Nos expériences ont montré que cette modification est très intéressante. Non seulement le nombre de chemins diminue (environ 10^{16}), mais la proportion de chemins infaisables aussi. Ainsi, pour la recherche d'un jeu de tests de taille 850, le nombre de chemins infaisables tirés est en moyenne de moins de 10^3. Pour générer 850 jeux de test couvrant le critère "tous les enchaînements" du programme FCT4, il nous faut entre deux et trois heures contre une semaine lors de nos premiers essais (cf section 7.5.2.1).

Les résultats du tableau 7.8 montrent les résultats obtenus pour cette expérience avec une probabilité minimale p_{min} de 0.3323 et une qualité de test q_N de 0.9999. Le score moyen de mutation est de 0.9773 avec un écart type de 0.0059.

Nous avons constaté que 4 jeux sur 5 n'ont tiré aucun chemin faisable passant

FCT4	score de mutation
série 1	0.9726
série 2	0.9726
série 3	0.9854
série 4	0.9726
série 5	0.9835

Tab. 7.8 : *Résultats pour FCT4 pour l'expérience 2*

par la branche *Else* de l'instruction *IfThenElse* de condition

$$SE\_VOIE\_EN\_TEST = TRUE$$

Le dernier jeu a tiré quant à lui un seul chemin faisable passant par cette branche *Else*.

De nouvelles expériences combinant la modification de la structure combinatoire et la première stratégie proposée, c'est-à-dire celle consistant à ne pas refaire de tirage dans l'ensemble S, n'ont guère été plus efficaces.

En fait, les chemins amenant à l'élimination de ces mutants non équivalents restent trop peu nombreux :

- 687×10^9 chemins passent par la branche où la condition (SE_VOIE_EN_TEST=TRUE) est fausse
- 10^{15} chemins passent par la branche où la condition (SE_VOIE_EN_TEST=TRUE) est vraie

Les résultats pour ce programme sont comparables à ceux obtenus par le LAAS.

7.5.2.3 Troisième expérience

Cette troisième expérience a consisté à modifier le code de FCT4 (sans le code mort) d'une manière telle que la spécification combinatoire utilisée dans l'expérience précédente corresponde (sans modification outre la contrainte sur le nombre d'itération des boucles). Ce programme FCT4′ est équivalent à FCT4 mais contient donc deux boucles et du code dupliqué.

Dans ce cas, pour pouvoir évaluer notre approche sur ce "nouveau" programme, nous avons dû modifier et élaborer de nouveaux mutants (duplication de code) de FCT4. Le nombre de mutants est passé de 605 à 724.

Les résultats des jeux de tests sont très disparates d'un jeu à l'autre. Le tableau 7.9 récapitule ces résultats. Le score moyen de 0.8804 avec un écart type de 0.0497.

FCT4′	score de mutation
série 1	0.8398
série 2	0.8398
série 3	0.9420
série 4	0.8398
série 5	0.9406

Tab. 7.9 : *Résultats pour FCT4′ pour l'expérience 3*

Comme précédemment, 4 jeux sur 5 n'assurent pas la couverture des éléments. En fait, on retrouve le même phénomène que l'expérience 2 : trop peu de chemins permettent d'obtenir la condition $SE\_VOIE\_EN\_TEST = TRUE$ fausse. Ce phénomène est même amplifié par la présence des nouveaux mutants : si aucun chemin ne permet d'avoir cette condition fausse alors tout programme dont la mutation dépend de cette condition fausse reste vivant.

7.5.3 Les expériences sur FCT4 et la programmation linéaire

Cette section présente les expériences réalisées à l'aide de la deuxième version du prototype, c'est-à-dire celle dont la distribution sur les éléments est définie à partir d'un programme linéaire. Nous avons réalisé les trois expériences précédentes.

7.5.3.1 Première expérience

Pour cette expérience, la distribution calculée donne une probabilité $\pi_{e0} = 1$ à l'enchaînement $e0$. Cet enchaînement correspond à l'arc qui sort du sommet INIT dans le graphe de la figure 7.1 : l'ensemble des chemins passant par cet enchaînement est l'ensemble de tous les chemins du graphe. Couvrir "tous les enchaînements" de ce programme en optimisant la qualité de test revient à tirer uniformément des chemins de ce graphe.

En effet, étant donné le nombre de chemins (près de 10^{20}) qui passent exactement 18 ou 19 fois dans la boucle, si l'on fait du tirage uniforme parmi tous les chemins alors les enchaînements contenus dans la boucle seront couverts avec une probabilité de 0.8 et les enchaînements liés aux sommets $NB\_VOIES = 18$ et $NB\_VOIES = 19$ seront couverts avec une probabilité de $\frac{1}{2}$. La probabilité minimale p_{min} d'atteindre un enchaînement ne peut être dans ce graphe qu'au maximum de $\frac{1}{2}$.

Pour le graphe de la figure 7.1, et intuitivement pour tous les graphes équilibrés, les critères "tous les enchaînements" et "toutes les instructions" semblent se ra-

mener à faire du tirage uniforme parmi tous les chemins du graphe.

Cette expérience a donc consisté à tirer 850 chemins faisables parmi un ensemble de 10^{20} chemins dont plus de 99% sont infaisables. La génération d'un premier jeu de test a demandé également une semaine. Le score de mutation obtenu par ce jeu est de 0.9726.

7.5.3.2 Deuxième et troisième expériences

Pour cette deuxième expérience, la distribution calculée a été la suivante (voir la figure 7.3 pour l'annotation des enchaînements) :

$$\begin{cases} \pi_{e11} = \pi_{e14} = \pi_{e17} = \pi_{e22} = \pi_{e25} = \pi_{e27} = 0.0844 \\ \pi_{e38} = \pi_{e39} = \pi_{e42} = \pi_{e45} = 0.1235 \\ \text{et } \pi_x = 0 \text{ pour tous les autres enchaînements} \end{cases}$$

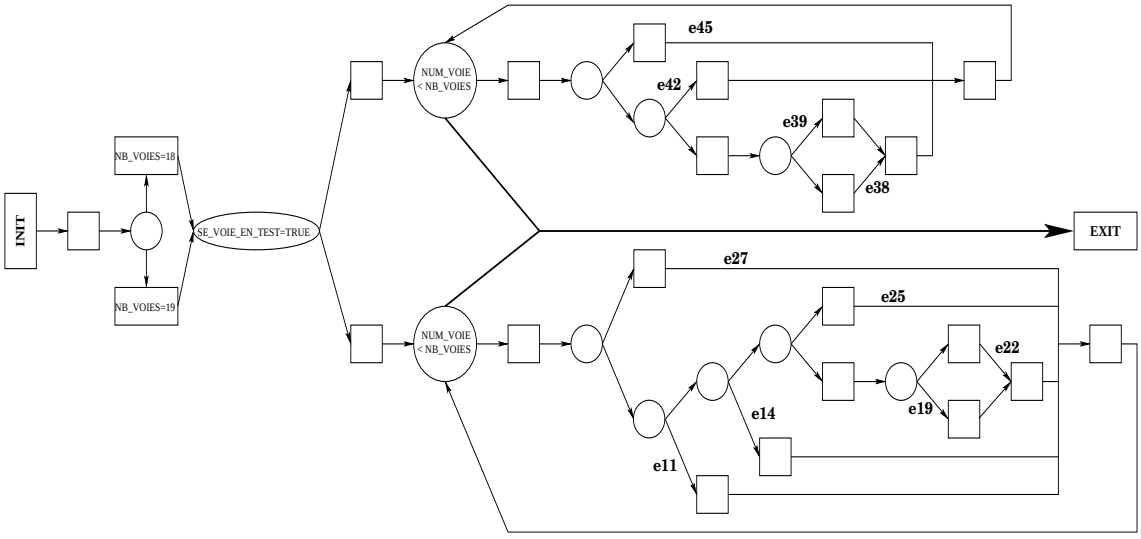


Fig. 7.3 : *Graphe de contrôle modifié, annoté et du programme FCT4 (sans code mort)*

Les résultats obtenus avec cette distribution sont bien meilleurs qu'avec l'ancienne (cf. tableaux 7.8 et 7.10) : non seulement le score de mutation moyen de 0.9854 est meilleur mais la couverture des éléments est toujours assurée. En mettant des probabilités π plus importantes aux enchaînements de la plus petite boucle c'est-à-dire celle par laquelle passe le moins de chemins, l'équilibre est rétabli.

De la même manière que pour l'expérience précédente, les résultats de l'expérience 3 sont bien meilleurs du point de vue du score de mutation (cf. tableaux 7.11 et 7.9) comme du point de vue de la couverture des éléments.

Le temps total de génération des tests est plus long qu'avec la distribution précédente. Le nombre de structures combinatoires à construire est beaucoup plus important et coûte deux jours de calculs (contre une heure). Cependant une fois la distribution et les structures combinatoires construites, les tests sont également générés en deux heures.

FCT4	score de mutation
série 1	0.9854
série 2	0.9854
série 3	0.9854
série 4	0.9854
série 5	0.9854

Tab. 7.10 : *Nouveaux résultats pour FCT4 pour l'expérience 2*

FCT4′	score de mutation
série 1	0.9878
série 2	0.9878
série 3	0.9878
série 4	0.9878
série 5	0.9878

Tab. 7.11 : *Nouveaux résultats pour FCT4′ pour l'expérience 3*

7.6 Bilan

Les expériences présentées dans ce chapitre ont été réalisées en deux phases : D'abord en utilisant la première version d'AuGuSTe pour les quatre programmes puis en utilisant la deuxième version uniquement sur le dernier programme car c'est le seul auquel on a appliqué un autre critère que celui de "tous les chemins".

Ces expériences avaient pour objectif d'évaluer notre approche sur une étude de cas pratique et de la comparer à une méthode de test existante dont nous connaissions les résultats. Nous n'avons présenté dans ce chapitre que les expériences utilisées pour cette comparaison. Le lecteur trouvera dans l'annexe B d'autres expériences (encore en cours) et nos premiers commentaires sur leurs résultats.

Le tableau 7.12 récapitule les résultats que nous avons obtenus et présentés dans ce chapitre. Pour l'ensemble des programmes, notre approche s'est montrée comparable à celle du LAAS. Cependant, notre processus de génération de test, de

	♯mutants non équivalents	score de mutation	♯mutants non tués
FCT1	265	1	0
FCT2	557	1	0
FCT3	1430	min=0.999 moy=0.9951 max= 1	entre 0 et 7
FCT4 (1)	547	min=0.9634 moy=0.9762 max=0.9854	entre 8 et 20
FCT4 (2)	547	min=0.9854 moy=0.9854 max=0.9854	8

Tab. 7.12 : *Résultats pour le test statistique structurel obtenus par AuGuSTe*

la construction de la distribution jusqu'aux tirages, est complètement automatisé quelque soit le programme testé contrairement au LAAS, où dans certains cas, la distribution doit être déterminée de manière empirique. De plus, parmi nos expériences complémentaires, présentées en annexe B, nous avons obtenu des jeux de test dont le score de mutation est parfait pour ce programme.

Comme nous l'avions déjà vu au chapitre 4, la présence des chemins infaisables semble déterminante. Pour pouvoir améliorer notre approche, il va falloir étudier comment les prendre en compte : que ce soit de manière statique (comme nous l'avons fait pour FCT4), de manière interactive (tests guidés par l'utilisateur), de manière probabiliste (ajout d'un taux d'erreur dans les calculs) ou par l'apprentissage.

Globalement, ces expériences ont donc été positives. Elles ont montré que :

- une automatisation du test statistique structurel était possible ;
- notre approche peut être utilisée pour des programmes réalistes : pour FCT4, les tirages ont été effectués dans un ensemble d'environ 10^{20} chemins et les prédicats associés aux chemins tirés comportaient en moyenne 190 conjonctions.

Plus généralement, elles justifient l'intérêt de continuer nos recherches dans cette voie.

Conclusion

Dans cette thèse, nous avons proposé une nouvelle méthode de test statistique, complètement automatisée. Notre approche est basée sur l'utilisation de structures combinatoires et des outils qui permettent de les manipuler. Nous avons vu que les descriptions sous forme de graphes de systèmes à tester peuvent généralement être modélisées par de telles structures.

À partir de ces structures combinatoires, d'un critère de couverture et d'une qualité de test (ou d'un nombre de tests) désirée, notre méthode permet de tirer aléatoirement un ensemble de chemins. Cet ensemble permet de couvrir tous les éléments du critère avec une certaine probabilité, en optimisant la probabilité minimale qu'un élément a d'être atteint.

Cette approche a été appliquée au test statistique structurel. Son implémentation par le prototype AuGuSTe a permis une première comparaison avec une méthode similaire développée au LAAS. Les expériences effectuées ont montré que notre approche était comparable à celle du LAAS. Même si les résultats ne sont pas toujours aussi bons que ceux du LAAS, la méthode s'avère stable et a l'avantage d'être complètement automatisée. Ces résultats nous encouragent à continuer nos travaux dans cette voie.

La complexité de la structure de certains programmes, comme le programme FCT4 du chapitre 7, a pour effet d'augmenter énormément le nombre de chemins considérés et en particulier le nombre de chemins infaisables. Pour réduire l'ensemble de ces chemins, nous avons proposé de modifier la structure combinatoire du programme FCT4 en utilisant les caractéristiques du programme (exemple : nombres fixés d'itérations de la boucle) mais sans modifier directement ce dernier.

Les résultats expérimentaux nous encouragent à développer notre approche et à travailler ses points faibles. Nous présentons ici quelques orientations possibles pour nos travaux futurs.

Poursuivre des expérimentations

Une première étape, en cours, consiste à valider expérimentalement la deuxième version d'AuGuSTe dont, théoriquement, la méthode de sélection des chemins est améliorée pour augmenter la qualité de test. Les premiers résultats obtenus sont comparés à ceux du LAAS mais également à ceux obtenus avec la première version d'AuGuSTe. Bien que plus efficace, la méthode calculant la distribution à partir d'un programme linéaire se heurte aux mêmes problèmes de couverture des chemins que celle basée sur les dominances et est plus coûteuse à construire. Une étude pour réduire ce coût est donc nécessaire (cf. section suivante).

Après ces expériences sur le test statistique structurel, il nous faudra également évaluer notre approche en l'appliquant au test fonctionnel. Cette évaluation se fera certainement de manière identique aux expériences du chapitre 7. En effet, la méthode du LAAS a également été développée pour le test statistique fonctionnel et les résultats d'expériences sont disponibles [117].

Améliorer le calcul des α_{ij}

Comme nous l'avons déjà dit, la méthode calculant la distribution à partir d'un programme linéaire est plus efficace mais aussi plus coûteuse à construire que celle basée sur les dominances.

En effet, pour le calcul des différents α_{ij} (nombre de chemins passant par l'élément e_i et l'élément e_j), elle nécessite la construction de $\frac{|E_C(D)|^2+|E_C(D)|}{2}$ automates contre, au pire, $|E_C(D)|$ automates pour la méthode basée sur les dominances. Pour réduire le nombre de ces automates, nous étudions le moyen d'utiliser la notion de dominance[5] et d'équivalence[6] entre éléments pour identifier les α_{ij} équivalents. Ainsi si e_i et e_j sont deux éléments équivalents alors $\alpha_{ij} = \alpha_i = \alpha_j$ et si e_i domine e_j alors $\alpha_{ij} = \alpha_j$.

De plus, les automates A_{e_i,e_j} des chemins passant par e_i et e_j sont actuellement construits par intersection à partir de A_{e_i} et A_{e_j}. Malheureusement, ces automates de départ peuvent être gros : le plus petit automate de FCT4 contient plus de 500 états. Leur intersection est alors souvent impraticable à calculer car elle amène rapidement à une explosion du nombre d'états de l'automate résultat. Pour remédier à ce problème, nous étudions deux solutions :

- Calculer l'automate A_{e_i,e_j} par l'intersection de A_{e_i} avec l'automate reconnaissant le langage $\mathcal{A}^* e_j \mathcal{A}^*$. L'automate résultat a au plus 2 fois le nombre d'états de A_{e_i} ;

[5] e_i domine e_j si tous les chemins qui passent par e_j passent par e_i.

[6] e_i et e_j sont équivalents si tous les chemins qui passent par e_i passent par e_j et réciproquement.

– Remplacer le calcul de l'automate A_{e_i,e_j} par celui de l'automate $A_{e_i,\bar{e}_j}$ des chemins passant par e_i mais pas par e_j. Cet automate contient au plus le même nombre d'états que A_{e_i}. Le calcul du nombre de chemins passant par e_i et e_j devient alors $\alpha_{e_i,e_j} = \alpha_{e_i} - \alpha_{e_i,\bar{e}_j}$.

Un graphe peut être représenté par différentes structures combinatoires. Bien que ces structures représentent le même ensemble de mots, le coût en temps de calcul de la phase de dénombrement peut être différent. C'est pourquoi, nous allons étudié comment modifier la traduction d'un graphe afin d'obtenir une structure combinatoire "optimisée" et ainsi réduire le temps de calcul.

Optimiser la distribution

Nous avons vu au chapitre 4 que la distribution pouvait être biaisée par des chemins infaisables ou par une couverture incomplète de tous les chemins faisables. Cela arrive lorsque cette distribution porte sur les éléments de $E_C(D)$ qui ne sont pas des chemins.

En effet, dans les cas où ce sont des chemins, en utilisant la méthode aléatoire de rejet, tirer dans un ensemble contenant des chemins infaisables revient à tirer uniformément dans un ensemble ne contenant que des chemins faisables. De plus, tous les chemins faisables sont tirables puisque la distribution est uniforme sur tous les chemins.

Dans le cas où les éléments ne sont pas des chemins, nous pensons améliorer le traitement des chemins infaisables en utilisant des techniques d'inférence et d'apprentissage ou des techniques probabilistes.

Les techniques d'inférence et d'apprentissage pourraient permettre l'adaptation de notre distribution en fonction des tirages effectués et des chemins infaisables identifiés. Par exemple, à chaque cycle tirage-résolution, des chemins infaisables pourraient être enlevés de l'ensemble des chemins considérés et la distribution recalculée à partir de ce nouvel ensemble. Un premier contact avec les membres de l'équipe "Inférence et Apprentissage" du LRI a déjà eu lieu et une première étude a été lancée. Ces travaux ont pour objectif l'identification de chemins infaisables par apprentissage à partir de l'ensemble de tous les chemins du graphe, d'un sous-ensemble de chemins faisables et d'un sous-ensemble de chemins infaisables.

Quant aux techniques probabilistes, elles pourraient permettre d'ajouter un certain "taux d'erreur" (exprimant par exemple la probabilité qu'un chemin soit infaisable) lors du calcul de la distribution. Le taux d'erreur étant soit dynamique c'est-à-dire calculé en fonction des tirages déjà effectués et du nombre de chemins infaisables trouvés, et dans ce cas une nouvelle distribution est calculée ; soit statique c'est-à-dire évalué à partir de mesures de complexité du programme.

Concernant le problème de couverture des chemins faisables qui pourraient être exclus par la distribution, nous pensons que toute solution choisie doit être a priori adaptée au contexte du test (critère, nombre et caractéristiques de ces chemins, etc.). Nous rappelons les propositions faites au chapitre 4 :

- ignorer les chemins écartés qui sont infaisables ou ne permettent pas de couvrir un des éléments de $E_C(D)$;
- considérer ces k chemins comme étant tirés, puis tirer $N - k$ chemins selon la distribution calculée ;
- fixer une probabilité d'être tiré non nulle (mais la plus petite possible) à un élément particulier et recalculer la distribution ; puis à chaque fois que cet élément sera tiré, effectuer un tirage parmi tous les chemins.

Étant donné que ces problèmes ne se posent que dans le cas où les éléments de $E_C(D)$ ne sont pas des chemins, on peut se demander si le choix (cf. section 4.2.1) de découper l'étape de tirage en deux étapes a été judicieux. Cependant, il reste pour l'instant le choix le plus simple. Une autre solution consistant par exemple à calculer directement une distribution sur les chemins permettant de couvrir avec une bonne qualité les éléments de $E_C(D)$ a été écartée car peu réaliste voire impossible quand il y a une infinité de chemins.

Orienter la distribution

Une évolution possible de notre approche pourrait être d'autoriser la modification de la distribution afin de prendre en compte certains cas particuliers. En effet, considérons le code suivant qui particularise deux valeurs `a` et `b` :

```
if(x=b)
then write("erreur")
else
  if (x=a)
  then traitement spécifique à a
  else traitement général
```

Est-ce vraiment intéressant de tester plusieurs fois le cas où la condition $x = b$ (resp. $x = a$) vaut vrai ? Ne peut-on pas se contenter de les tester un nombre de fois fixé ? Par exemple, un seul test pour $x = b$ semble suffisant puisque le traitement du cas associé est très simple. Si l'on s'intéresse maintenant au cas $x = a$, le traitement spécifique peut correspondre à un sous-graphe compliqué mais conceptuellement simple alors que le cas $x <> a$ peut correspondre à une unique expression (un unique chemin donc) mais de complexité réelle bien plus importante. Dans de tels cas, faut-il traiter tous les cas nécessaires pour $x = a$ ou ne pas les différencier ? Quant au cas général $x <> a$, faut-il le tester plusieurs fois ?

La distribution pourrait donc être orientée, par exemple, à l'aide de métriques sur le programme (mesures d'Halstead [67] ou autres) ou d'indications sémantiques ajoutées au code. Le prototype AuGuSTe pourrait également autoriser l'utilisateur à définir ses propres critères en lui permettant par exemple de choisir, à l'aide d'une interface, des éléments (sommets et/ou arcs) du graphe de contrôle. Une étude plus générale de ce problème est envisagée.

Extensions possibles de notre approche

Cette section présente diverses idées, à plus long terme, que nous aimerions également étudier pour étendre notre approche.

Une première idée porte sur l'exploitation des résultats théoriques de la génération aléatoire basée sur le modèle de Boltzmann (présentée dans la section 3.1.2) qui pourrait éventuellement amener au tirage de chemins de longueur non bornée.

Ensuite, nous aimerions exploiter toutes les possibilités offertes par les packages CS et MuPAD-Combinat. Notre approche pourrait ainsi être étendue à des langages plus complexes que les langages réguliers : la spécification combinatoire pourrait exprimer des propriétés de la description comme par exemple, le fait que deux boucles du graphe soient itérées exactement le même nombre de fois ou sont exclusives l'une de l'autre. Cette idée pourrait également être utilisée pour limiter, voire éliminer, les chemins infaisables.

Pour terminer, les résultats expérimentaux obtenus pour des programmes non triviaux nous permettent d'envisager un passage à l'échelle de notre approche et par conséquent le développement d'un outil destiné au monde industriel. De plus, ces travaux pourraient servir de base pour une nouvelle classe d'outils dans le domaine du test de logiciel.

Annexe A

Grammaire des programmes considérés

$token < int > INT$
$token < float > FLOAT$
$token < bool > BOOL$
$token < string > VAR$

$$
\begin{array}{lcl}
source & := & VAR\, parametres\{argumentsDefinition; suiteInstructions\} \\
| & & VAR\, parametres\{suiteInstructions\}
\end{array}
$$

$$
\begin{array}{lcl}
parametres & := & (parametreDefinition) \\
| & & ()
\end{array}
$$

$$
\begin{array}{lcl}
parametreDefinition & := & VAR, typ \\
| & & parametreDefinition, VAR : typ
\end{array}
$$

$$
\begin{array}{lcl}
typ & := & TYPEINT \\
| & & TYPEVOID \\
| & & TYPEFLOAT \\
| & & TYPEBOOL \\
| & & TYPESTRING \\
| & & VAR \\
| & & ARRAY\ INT\ INT\, typ
\end{array}
$$

$$
\begin{array}{lcl}
argumentsDefinition & := & variableLocale \\
| & & argumentsDefinition; variableLocale
\end{array}
$$

$$
\begin{array}{lcl}
variableLocale & := & VAR : typ
\end{array}
$$

$$
\begin{array}{lcl}
suiteInstructions & := & instructions; suiteInstructions \\
| & & instructions \\
 & & \\
instructions & := & affectation \\
| & & specs \\
| & & IF(expr)\,THEN\{suiteInstructions\} \\
| & & FOR(affectation; expr; affectation) \\
 & & \quad \{suiteInstructions\} \\
| & & WHILE(expr)[liste\_conditions]\{suiteInstructions\} \\
| & & WHILE(expr)\{suiteInstructions\} \\
| & & RETURN(expr) \\
| & & IF(expr)\,THEN\{suiteInstructions\} \\
 & & \quad ELSE\{suiteInstructions\} \\
 & & \\
affectation & := & VAR\ AFFECT\ expr \\
| & & VAR[expr]AFFECT\ expr \\
 & & \\
specs & := & SPEC\ EQUAL(listeSpec) \\
 & & \\
listeSpec & := & spec, listeSpec \\
| & & spec \\
 & & \\
spec & := & VAR\ IN\ [borne, borne] \\
| & & VAR\ [expr]\ IN\ [borne, borne] \\
| & & expr \\
 & & \\
borne & := & varint \\
| & & MINUS\ varint \\
 & & \\
liste\_conditions & := & condition \\
| & & condition; liste\_conditions \\
 & & \\
condition & := & INT \\
| & & GREATER\ INT \\
| & & LESS\ INT \\
 & & \\
varint & := & INT \\
| & & VAR \\
| & & MAXINT \\
| & & MININT \\
 & & \\
expr & := & expr\ OUEXCLUSIF\ expr0 \\
| & & expr0
\end{array}
$$

$$
\begin{array}{lcl}
expr0 & := & expr0\ OR\ expr1 \\
| & & expr1 \\
 & & \\
expr1 & := & expr1\ AND\ expr2 \\
| & & expr2 \\
 & & \\
expr2 & := & NOT\ expr3 \\
| & & expr3 \\
 & & \\
expr3 & := & expr3\ DBEQ\ expr4 \\
| & & expr4 \\
 & & \\
expr4 & := & expr4\ DISEQUAL\ expr5 \\
| & & expr5 \\
 & & \\
expr5 & := & expr5\ LEQ\ expr6 \\
| & & expr6 \\
 & & \\
expr6 & := & expr6\ GEQ\ expr7 \\
| & & expr7 \\
 & & \\
expr7 & := & expr7\ GREATER\ expr8 \\
| & & expr8 \\
 & & \\
expr8 & := & expr8\ LESS\ expr9 \\
| & & expr9 \\
 & & \\
expr9 & := & expr9\ PLUS\ expr10 \\
| & & expr10 \\
 & & \\
expr10 & := & expr10\ MINUS\ expr11 \\
| & & expr11 \\
 & & \\
expr11 & := & expr11\ TIMES\ expr12 \\
| & & expr12 \\
 & & \\
expr12 & := & expr12\ DIV\ expr13 \\
| & & expr13 \\
 & & \\
expr13 & := & (expr) \\
| & & BOOL \\
| & & varint \\
| & & FLOAT \\
| & & VAR[expr] \\
| & & MINUS\ expr13
\end{array}
$$

Annexe B

Résultats d'expériences

Cette annexe présente un exemple d'expériences supplémentaires que nous avons réalisées. Le premier ensemble d'expériences est quantitatif : nous avons réalisé beaucoup plus d'expériences que celles présentées au chapitre 7. Le deuxième ensemble, dont l'analyse est encore en cours, a été réalisé afin de répondre à différentes interrogations comme quelle est l'influence de la qualité de test sur la couverture, en pratique, du critère, etc.

Nous présentons dans cette annexe toutes ces expériences et nos (premiers) commentaires.

B.1 Expériences supplémentaires réalisées

Lors de notre campagne d'expériences, nous avons réalisé en fait cinq fois plus d'expériences que le LAAS. Notre objectif était double : se comparer au LAAS et étudier en détails notre approche. Dans le chapitre 7, nous avons présenté les résultats obtenus par la ou les cinq première(s) série(s) de chaque expérience. C'est pourquoi les résultats présentés ne sont pas forcément les meilleurs. Nous présentons ici l'ensemble des résultats.

Les tableaux B.1 et B.2 montrent que tous les jeux de test obtenus pour les programmes FCT1 et FCT2 ont obtenu un score de mutation parfait.

Le tableau B.3 présente les scores de mutation obtenus pour le programme FCT3 par 25 jeux de tests : 2 jeux sur 3 ont obtenu un score parfait. Les autres ont laissé entre 1 et 14 mutants vivants. Ce qui nous donne pour le score de mutation une moyenne de 0.9981 avec un écart type de 0.0033. La stabilité de notre approche est très satisfaisante surtout pour ce type de programme où l'exécution des tests est très dépendante de l'environnement et en particulier de l'ordre dans lequel on exécute ces tests.

FCT1	score de mutation
série 1	1
série 2	1
série 3	1
série 4	1
série 5	1

Tab. B.1 : *Résultats pour FCT1*

FCT2	score de mutation
série 1	1
série 2	1
série 3	1
série 4	1
série 5	1

Tab. B.2 : *Résultats pour FCT2*

score de mutation	♯mutants non tués	♯jeux concernés
1	0	17
0.9993	1	1
0.9986	2	1
0.9944	8	3
0.9909	13	1
0.9902	14	2

Tab. B.3 : *Résultats pour FCT3*

score de mutation	♯mutants non tués	♯jeux concernés	critère couvert
1	0	1	oui
0.9963	2	2	oui
0.9890	6	1	non
0.9872	7	1	oui
0.9854	8	3	oui
0.9854	8	1	non
0.9835	9	4	oui
0.9726	15	11	non
0.9598	22	1	non

Tab. B.4 : *Résultats obtenus pour FCT4(1)*

score de mutation	♯mutants non tués	♯jeux concernés
1	0	1
0.9963	2	2
0.9890	6	7
0.9854	8	12
0.9835	9	1
0.9762	13	1
0.9726	15	1

Tab. B.5 : *Résultats obtenus pour FCT4(2)*

Les expériences supplémentaires présentées pour FCT4 sont basées sur l'expérience 2 (cf. section 7.5.2.2, page 133). Dans la suite, nous utilisons la notation FCT4(1) et FCT4(2) lorsque nous faisons référence à des expériences effectuées sur FCT4 en utilisant respectivement la distribution calculée à l'aide des dominances ou celle calculée à l'aide du programme linéaire.

Le tableau B.4 présente les résultats obtenus pour FCT4 par notre approche utilisant une distribution calculée à l'aide des dominances. Le score de mutation moyen est de 0.9801 pour un écart type de 0.0095. 14 jeux sur 25 n'assurent pas la couverture du critère : ce sont ceux qui ont généralement les plus mauvais scores de mutation. Cependant quatre jeux ont obtenu le même score de mutation 0.9854 et un des quatre n'assure pas la couverture du critère. On remarque également que dans cette série de 25 jeux de longueur 850, un jeu a permis de tuer tous les mutants non équivalents.

Le tableau B.5 présente les résultats obtenus pour FCT4 par notre approche utilisant une distribution calculée à du programme linéaire. Le score de mutation moyen est de 0.9869 pour un écart type de 0.0054. Tous les jeux obtenus assurent la couverture des éléments. On remarque également que dans cette série de 25 jeux de longueur 850, un jeu a permis de tuer tous les mutants non équivalents.

B.2 Influence de la qualité de test

Ces expériences sont en cours d'exécution, c'est pourquoi certains résultats restent partiels.

Nous avons voulu étudier l'influence de la qualité de test sur la couverture obtenue en pratique. Pour les programmes FCT1 et FCT4 (avec les deux distributions), nous avons calculé le nombre de tests nécessaires pour couvrir un critère avec quatre qualité de tests : 0.9, 0.99, 0.999 et 0.9999. Le tableau B.6 récapitule

pour ces trois programmes le nombre de test N calculées. On peut remarquer que pour couvrir les 17 chemins de FCT1, il faut plus de tests que pour couvrir les 41 enchaînements de FCT4. Cela s'explique par le fait que le p_{min} de FCT1 est de 0.0588 et donc nettement inférieur aux p_{min} calculés pour FCT4 qui sont respectivement 0.3323 pour FCT4(1) et 0.49 pour FCT4(2).

	q_N	0.9	0.99	0.999	0.9999
FCT1	N	38	76	114	152
FCT4(1)	N	6	12	18	23
FCT4(2)	N	4	8	11	14

Tab. B.6 : *Nombre de tests N nécessaire pour $q_N \in \{0.9, 0.99, 0.999, 0.9999\}$*

Ensuite, nous avons généré pour chaque programme, 100 jeux de taille N tels que $q_N \in \{0.9, 0.99, 0.999, 0.9999\}$. Puis nous avons observé la couverture obtenue en pratique ainsi que le score moyen de mutation obtenu. Les tableaux B.7, B.8 et B.9 présentent les taux de couverture constatés en pratique pour FCT1, FCT4(1) et FCT4(2). Pour FCT1, la couverture des chemins est assurée à partir d'une qualité de test de 0.9999.

Pour FCT4, aucune des deux distributions n'assure la couverture des enchaînements pour cette qualité mais on peut remarquer que les taux de couverture évoluent différemment pour les deux. En effet, pour FCT4(1), la couverture est loin d'être atteinte alors que la distribution utilisée pour FCT4(2), qui nécessite moins de tests pour une qualité donnée, obtient un taux de couverture de 85%. Les résultats de FCT4(1) ne sont pas surprenants puisque les expériences précédentes avaient montré que pour 25 jeux de longueur de test $N = 850$, 14 n'avaient pas couvert le critère. Quant à FCT4(2), nos expériences ont montré qu'à partir de $N = 41$ soit $q_{41} = 1 - 10^{-12}$, la distribution permet d'obtenir à tous les coups la couverture des éléments.

Les tableaux B.10, B.11 et B.12 présentent les scores de mutation obtenus par les différentes séries de jeux de test. En observant le tableau B.10, on peut remarquer que même si un jeu de test assure la couverture des chemins d'exécution, il ne permet pas toujours de tuer tous les mutants non équivalents.

Les résultats des tableaux B.11 et B.12, mais aussi B.4 et B.5, nous permettent, quant à eux, de conclure sur une première comparaison expérimentale des deux distributions. Du point de vue couverture des éléments d'un critère, la distribution calculée à l'aide de la programmation linéaire est la plus efficace pour un minimum de test. Par contre, du point de vue de l'efficacité dans la détection des erreurs, la différence entre les deux distributions n'est pas aussi tranchée : les résultats obtenus par FCT4(2) sont meilleurs mais pas nettement supérieurs à ceux obtenus par FCT4(1).

♯ chemins non couverts	♯ jeux concernés			
	pour $N = 38$ et $q_{38} = 0.9$	pour $N = 76$ et $q_{76} = 0.99$	pour $N = 114$ et $q_{114} = 0.999$	pour $N = 152$ et $q_{152} = 0.9999$
0	13	**85**	**98**	**100**
1	**40**	14	2	0
2	28	1	0	0
3	17	0	0	0
4	2	0	0	0

Tab. B.7 : *Taux de couverture constatés pour FCT1*

♯ enchaînements non couverts	♯ jeux concernés			
	pour $N = 6$ et $q_6 = 0.9$	pour $N = 12$ et $q_{12} = 0.99$	pour $N = 18$ et $q_{18} = 0.999$	pour $N = 23$ et $q_{23} = 0.9999$
0	0	0	0	1
18	**53**	**84**	**96**	**97**
20	47	16	4	2

Tab. B.8 : *Taux de couverture constatés pour FCT4(1)*

♯ enchaînements non couverts	♯ jeux concernés			
	pour $N = 4$ et $q_4 = 0.9$	pour $N = 8$ et $q_8 = 0.99$	pour $N = 11$ et $q_{11} = 0.999$	pour $N = 14$ et $q_{14} = 0.9999$
0	**47**	**74**	**87**	**87**
2	37	25	13	13
4	1	0	0	0
18	4	0	0	0
20	5	0	0	0
24	3	1	0	0
26	2	0	0	0

Tab. B.9 : *Taux de couverture constatés pour FCT4(2)*

nombre de tests	qualité de test	score de mutation		
		min	moy	max
$N = 38$	$q_{38} = 0.9$	0.9547	0.9685	0.9736
$N = 76$	$q_{76} = 0.99$	0.9547	0.9697	0.9698
$N = 114$	$q_{114} = 0.999$	0.9698	0.9978	1
$N = 152$	$q_{152} = 0.9999$	0.9698	0.9967	1

Tab. B.10 : *Scores de mutation constatés pour FCT1*

nombre de tests	qualité de test	score de mutation		
		min	moy	max
$N = 6$	$q_6 = 0.9$	0.7093	0.8135	0.9580
$N = 12$	$q_{12} = 0.99$	0.7130	0.8775	0.9616
$N = 18$	$q_{18} = 0.999$	0.7258	0.9082	0.9762
$N = 23$	$q_{23} = 0.9999$	0.7367	0.9194	0.9726

Tab. B.11 : *Scores de mutation constatés pour FCT4(1)*

nombre de tests	qualité de test	score de mutation		
		min	moy	max
$N = 4$	$q_4 = 0.9$	0.4845	0.8138	0.9707
$N = 8$	$q_8 = 0.99$	0.6636	0.8658	0.9634
$N = 11$	$q_{11} = 0.999$	0.7203	0.8958	0.9744
$N = 14$	$q_{14} = 0.9999$	0.7313	0.8917	0.9634

Tab. B.12 : *Scores de mutation constatés pour FCT4(2)*

B.3 Graphes de contrôle des FCTn

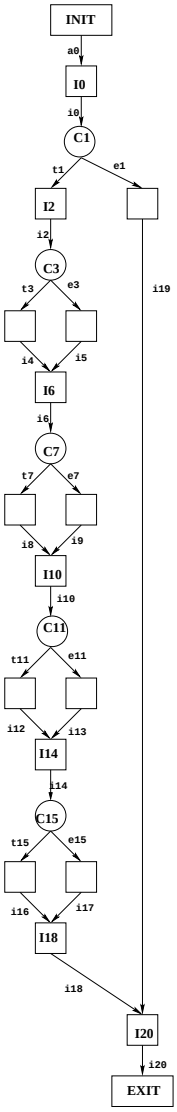


Fig. B.1 : *Graphe de contrôle de FCT1*

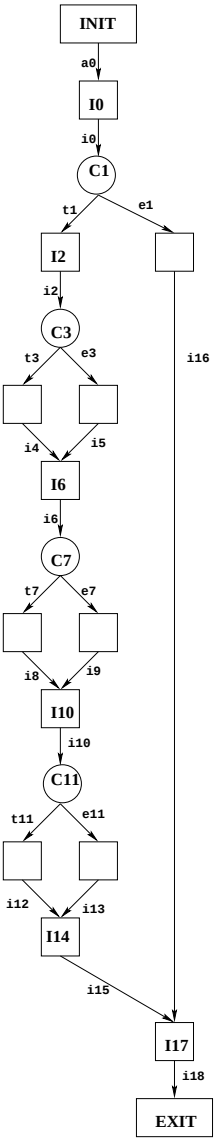


Fig. B.2 : *Graphe de contrôle de FCT2*

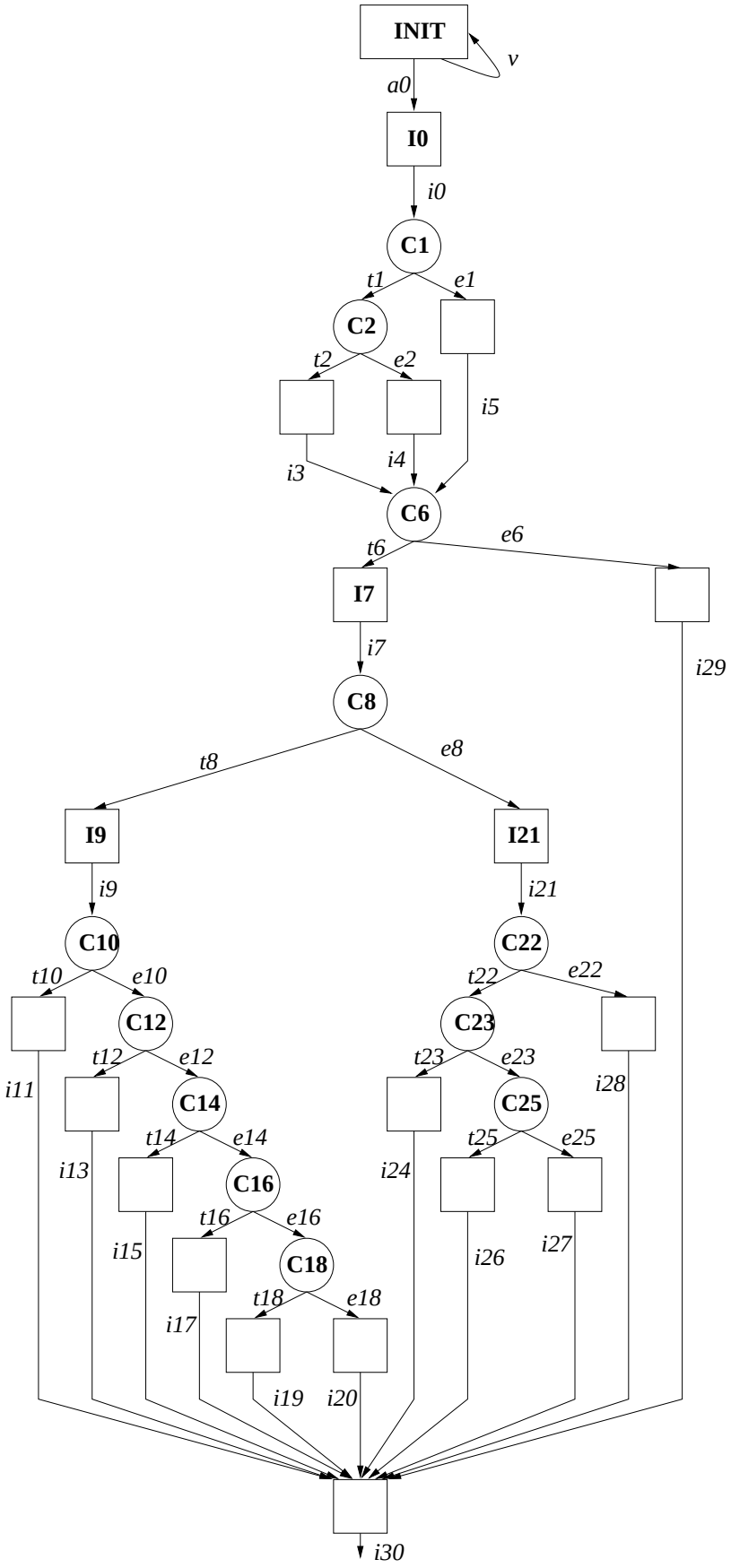


Fig. B.3 : *Graphe de contrôle de FCT3*

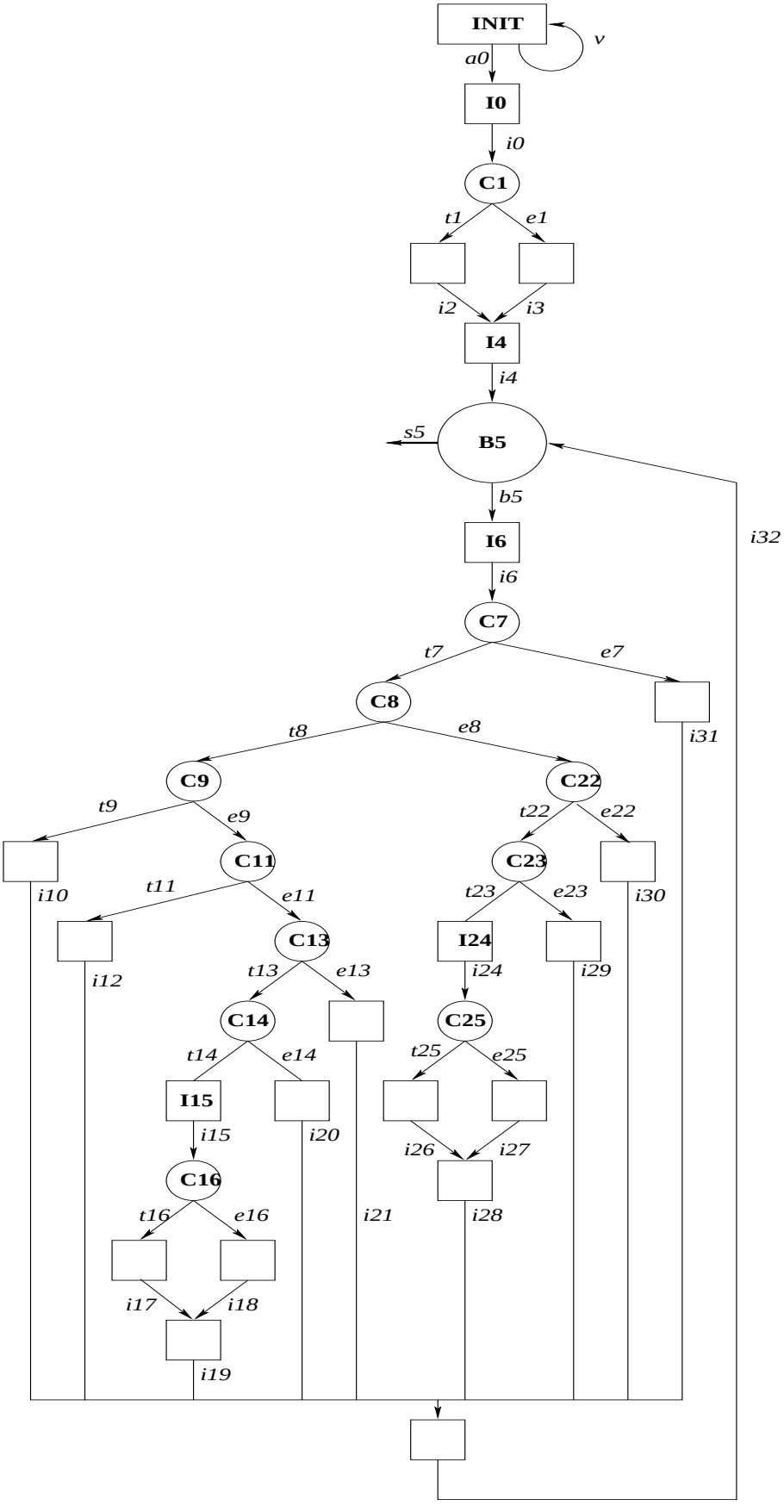


Fig. B.4 : *Graphe de contrôle de FCT4*

Annexe C

Algorithmes de base de la résolution de contraintes

Les techniques de résolution des CSP sont basées sur les techniques de consistance c'est-à-dire une propagation des valeurs avec réduction des domaines de variables et peuvent se classer en trois catégories [116] : celles qui se basent sur la simplification des problèmes, celles qui se basent sur la recherche de solutions et celles qui se basent sur la synthèse des solutions possibles. Nous présentons brièvement dans cette annexe ces trois catégories.

C.1 Simplifier le problème (*Problem reduction*)

Cette classe de techniques consiste à transformer un CSP en un problème plus facile à résoudre ou que l'on sait insatisfiable. Cette technique seule permet rarement de résoudre le problème mais peut être extrêmement utile pour les autres techniques.

Définition 17 (équivalence de CSP) *Deux CSP sont dits équivalents s'ils ont le même ensemble $\mathcal{X}$ de variables et les même ensembles I de solutions.*

Définition 18 (réduction de CSP)
Un CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) se réduit en un CSP($\mathcal{X}\prime, \mathcal{D}\prime, \mathcal{C}\prime$) si :

1. *CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) et CSP($\mathcal{X}\prime, \mathcal{D}\prime, \mathcal{C}\prime$) sont équivalents*
2. *pour chaque variable x de $\mathcal{X}$, son domaine $D\prime_x$ de $\mathcal{D}\prime$ est un sous-ensemble de son domaine D_x de $\mathcal{D}$.*
3. *$\mathcal{C}\prime$ est aussi ou plus restrictive que $\mathcal{C}$, c'est-à-dire que toute instanciation partielle qui satisfait $\mathcal{C}\prime$ satisfait $\mathcal{C}$.*

Une manière de simplifier un problème est de retirer tout élément qui est inutile comme par exemple des valeurs de variables qui ne peuvent faire partie des solutions d'un problème.

Définition 19 (valeur redondante) *Une valeur dans un domaine est dite redondante si elle ne fait partie d'aucune solution.*

Le terme « redondant » est utilisé parce que l'élimination de telles valeurs n'affecte pas l'ensemble de solution du problème.

Définition 20 (affectation redondante) *Une affectation d'une contrainte est dite redondante si elle n'est pas une affectation partielle d'une solution du problème.*

Les techniques de simplification des problèmes transforme un CSP en un problème équivalent, mais beaucoup plus facile, en réduisant la taille des domaines et les contraintes du problème. Cela n'est possible que parce que les domaines et les contraintes dans les CSP sont spécifiés et que les contraintes peuvent être propagées.

La simplification de problème consiste en deux taches :

1. retirer les valeurs redondantes
2. renforcer les contraintes pour diminuer le nombre d'affectation qui les satisfassent

Si la simplification du problème ramène au moins un domaine de valeurs à l'ensemble vide, alors le problème est insatisfiable.

Définition 21 (problème minimal) *Un CSP est un* **problème minimal** *si aucun de ces domaines ne contient de valeur redondante et aucune contrainte ne contient d'affectation redondante.*

Le filtrage des domaines est basé sur des notions de consistances. Nous présentons dans l'exemple 39 une manière de réduire le problème CSP correspondant au cryptarithmétique "SEND+MORE=MONEY".

De nombreux algorithmes permettant de simplifier un problème CSP se basent sur des différentes notions de consistances [116]. Cependant la simplification de problème seule ne permet pas toujours de trouver une solution au problème CSP considéré, c'est pourquoi on est amené à la combiner avec les autres techniques et en particulier celle de la recherche de solutions.

Exemple 39 (Cryptarithmétique : SEND+MORE=MONEY)

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Soit le CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) suivant :

- $\mathcal{X} = \{S, E, N, D, M, O, R, Y, r_1, r_2, r_3, r_4\}$
- $\mathcal{D} = \{D_S, D_E, D_N, D_D, D_M, D_O, D_R, D_Y, D_{r_1}, D_{r_2}, D_{r_3}, D_{r_4}\}$ *avec*
  - $D_S = D_E = D_N = D_D = D_M = D_O = D_R = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - $D_{r_1} = D_{r_2} = D_{r_3} = D_{r_4} = \{0, 1\}$
- $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ *avec*
  - $c_1 :\ D + E = Y + 10 \times r_1$
  - $c_2 :\ r_1 + N + R = E + 10 \times r_2$
  - $c_3 :\ r_2 + E + O = N + 10 \times r_3$
  - $c_4 :\ r_3 + S + M = O + 10 \times r_4$
  - $c_5 :\ r_4 = M$
  - $c_6 :\ M \neq 0$
  - $c_7 :\ S \neq 0$
  - $c_8 :\ toutes\_differentes(S, E, N, D, M, O, R, Y)$

On regarde les contraintes une par une en commençant par les contraintes unaires puis binaires, ...,jusqu'à n-aire.

- $c_6 :\ M \neq 0$ *permet de réduire le domaine de* M *à* $D_M = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $c_7 :\ S \neq 0$ *permet de réduire le domaine de* S *à* $D_S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $c_5 :\ r_4 = M$ *seule l'affectation* $\{(M, 1), (r_4, 1)\}$ *peut satisfaire* c_5*, les domaines de* M *et de* r_4 *se réduisent alors à* $D_M = D_{r_4} = \{1\}$ *et la contrainte* c_4 *se réduit à* $r_3 + S = O + 9$
- $c_8 :\ toutes\_differentes(S, E, N, D, M, O, R, Y)$ *permet de réduire les domaines de* S*,* E*,* N*,* D*,* O*,* R *et* Y *de la façon suivante :*
  $D_S = \{2, 3, 4, 5, 6, 7, 8, 9\}$ *et*
  $D_E = D_N = D_D = D_O = D_R = D_Y = \{0, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $c_4 :\ r_3 + S = O + 9$ *seules les affectations* $\{(r_3, 0), (S, 9), (O, 0)\}$ *et* $\{(r_3, 1), (S, 8), (O, 0)\}$ *peuvent satisfaire* c_4*, les domaines de* S *et de* O *se réduisent alors à* $D_S = \{8, 9\}$ *et* $D_O = \{0\}$*, et les contraintes* c_4 *et* c_3 *se réduisent respectivement à* $r_3 + S = 9$ *et* $r_2 + E = N + 10 \times r_3$
- $c_8 :\ toutes\_differentes(S, E, N, D, M, O, R, Y)$ *permet de réduire les domaines de* E*,* N*,* D*,* R *et* Y *à*
  $D_E = D_N = D_D = D_R = D_Y = \{2, 3, 4, 5, 6, 7, 8, 9\}$
- $c_3 :\ r_2 + E = N + 10 \times r_3$ *seules les affectations*
  $\{(r_2, 1), (E, 2), (N, 3), (r_3, 0)\}$, $\{(r_2, 1), (E, 3), (N, 4), (r_3, 0)\}$,
  $\{(r_2, 1), (E, 4), (N, 5), (r_3, 0)\}$, $\{(r_2, 1), (E, 5), (N, 6), (r_3, 0)\}$,

$\{(r_2,1),(E,6),(N,7),(r_3,0)\}$, $\{(r_2,1),(E,7),(N,8),(r_3,0)\}$,
$\{(r_2,1),(E,8),(N,9),(r_3,0)\}$, $\{(r_2,0),(E,2),(N,2),(r_3,0)\}$,
$\{(r_2,0),(E,3),(N,3),(r_3,0)\}$, $\{(r_2,0),(E,4),(N,4),(r_3,0)\}$,
$\{(r_2,0),(E,5),(N,5),(r_3,0)\}$, $\{(r_2,0),(E,6),(N,6),(r_3,0)\}$,
$\{(r_2,0),(E,7),(N,7),(r_3,0)\}$, $\{(r_2,0),(E,8),(N,8),(r_3,0)\}$
et $\{(r_2,0),(E,9),(N,9),(r_3,0)\}$ *peuvent satisfaire* c_3*, le domaine de* r_3 *se réduit alors à* $D_{r_3} = \{0\}$ *et les contraintes* c_3 *et* c_4 *se réduisent respectivement en* $r_2 + E = N$ *et* $S = 9$

- $c_4 : S = 9$ *permet de réduire le domaine de* S *à* $D_S = \{9\}$
- c_8 $toutes\_differentes(S,E,N,D,M,O,R,Y)$ *permet de réduire les domaines de* E*,* N*,* D*,* O*,* R *et* Y *de la façon suivante :*
  $D_E = D_N = D_D = D_O = D_R = D_Y = \{2,3,4,5,6,7,8\}$

Sans combiner plusieurs contraintes ensembles et sans faire d'énumérations, il est impossible de résoudre ce problème. En effet, il suffirait de combiner la contrainte c_8 *à chaque contrainte traitée, et en particulier avec* c_3*, pour résoudre ce problème.*

La version minimale du problème "SEND+MORE=MONEY" est donc le CSP($\mathcal{X},\mathcal{D},\mathcal{C}$*) suivant :*

- $\mathcal{X} = \{S,E,N,D,M,O,R,Y,r_1,r_2,r_3,r_4\}$
- $\mathcal{D} = \{D_S,D_E,D_N,D_D,D_M,D_O,D_R,D_Y,D_{r_1},D_{r_2},D_{r_3},D_{r_4}\}$ *avec*
  - $D_E = D_N = D_D = D_R = D_Y = \{2,3,4,5,6,7,8\}$
  - $D_S = \{9\}$
  - $D_M = D_{r_4} = \{1\}$
  - $D_O = D_{r_3} = \{0\}$
  - $D_{r_1} = D_{r_2} = \{0,1\}$
- $\mathcal{C} = \{c_1,c_2,c_3,c_4\}$ *avec*
  - $c_1 : D + E = Y + 10 \times r_1$
  - $c_2 : r_1 + N + R = E + 10 \times r_2$
  - $c_3 : r_2 + E = N$
  - $c_4 : toutes\_differentes(S,E,N,D,M,O,R,Y)$

C.2 Rechercher les solutions (*Search*)

C.2.1 L'algorithme de base

L'algorithme de base pour la recherche de solutions est le *backtracking simple* ou algorithme de "simple retour-arrière"[1] qui est une stratégie générale de recherche largement utilisé pour la résolution de problèmes. Prolog utilise notamment cette algorithme pour répondre aux requêtes.

Dans le contexte des CSP, l'opération de base est de prendre une variable, de lui choisir une valeur telle qu'elle soit compatible avec les instances choisies précédemment. Dans la communauté de la programmation par contraintes, l'assignation d'une valeur a une variable est appelée *labeling*. Si la valeur choisie viole au moins une contrainte, c'est-à-dire que quelque soit l'affectation choisie contenant cette instance, il existe au moins une contrainte qui ne peut être satisfaite, alors, si c'est possible, une autre valeur est choisie.

Si toutes les variables sont instanciées, alors le problème est résolu. Si à une étape donnée, aucune valeur ne peut être assignée à une variable sans violer une contrainte, alors la dernière assignation est remise en compte et si c'est possible une nouvelle valeur est choisie sinon on remonte ainsi jusqu'à la première variable assignée.

Au final, soit une solution est trouvée soit toutes les combinaisons possibles d'instances ont été essayées et ont échoués. La figure C.2.1 résume l'algorithme de *backtracking* simple.

Soit n le nombre de variables, e le nombre de contraintes et a la taille du plus grand domaine d'un CSP. Comme il y a a^n combinaisons possibles de solutions, et pour chaque affectation totale candidate, toutes les contraintes doivent être vérifiées dans le pire des cas, la complexité en temps de l'algorithme de *backtracking* simple est de $O(a^n e)$.

Le stockage des domaines du problème requiers au plus $O(na)$ d'espace. L'algorithme ne requiert pas plus de $O(n)$ d'espace pour mémoriser temporairement l'affectation candidate. La complexité en espace de l'algorithme de *backtracking* simple est alors de $O(na)$.

La complexité en temps vue ci-dessus montre que l'efficacité de la recherche peut être améliorée si a peut être réduit. Ce qui peut être fait à l'aide de la technique de simplification de problème mentionnée précédemment.

[1] Le terme français étant rarement employé.

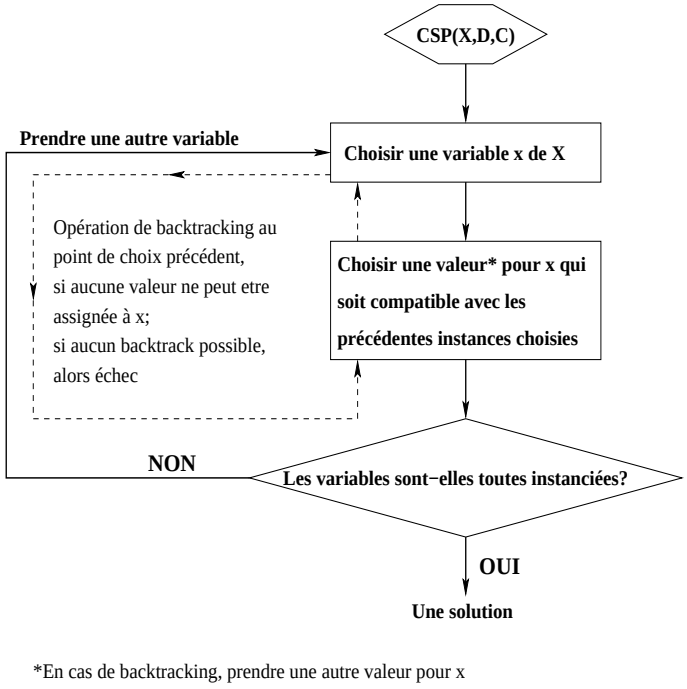


Fig. C.1 : *Schéma général de l'algorithme de backtracking simple*

C.2.2 Heuristiques sur les points de choix

C.2.2.1 Les trois points de choix de cette technique

Il y a trois points de choix dans l'algorithme de *backtracking* simple :

1. qu'elle sera le prochaine variable ?
2. qu'elle sera la prochaine valeur ?
3. qu'elle est la prochaine contrainte à regarder ?

En général, plus il y a de valeurs et d'instanciations à retirer, plus le coût de calcul est important. D'un autre coté, moins il y a de valeurs et d'instanciations à retirer, plus on passera du temps à *backtracker*. Il faut donc trouver un bon compromis entre les efforts à faire et le gain potentiel de la simplification du problème.

De plus, il y a autant d'arbre de recherche à explorer que d'ordonnancement des variables et de leurs valeurs. À partir du moment où les contraintes peuvent être propagées, l'ordre des variables et de leurs valeurs peut alors avoir une influence sur l'efficacité de l'algorithme de recherche. Cela est d'autant plus vrai que l'on utilise la simplification.

Pour les problèmes dont on attend qu'une seule solution, l'efficacité de la recherche peut être améliorer en utilisant des heuristiques qui permettent de

guider le choix des branches à explorer dans l'arbre de recherche.

Dans d'autres cas, vérifier si une contrainte est satisfiable ou non peut être un coût de calcul non négligeable. Alors, l'ordre dans lequel les contraintes vont être examinées peut affecter de manière significative l'efficacité d'un algorithme. Si la situation est sous-contrainte alors le plus tôt les contraintes violées sont examinées, le moins de calcul on effectuera.

C.2.2.2 L'espace de recherche

L'espace de recherche pour l'algorithme de *backtracking* simple est un arbre tel que les noeuds (resp. feuilles) sont les états où la recherche de solutions peut arriver c'est-à-dire toutes les affectations partielles (resp. affectations totales) possibles. Dans cet arbre, les contraintes ne jouent aucun rôle bien que comme nous le verrons plus loin, cela affecte l'exploration de l'espace de recherche par un algorithme.

L'arbre de recherche est un espace de recherche dans dans lequel on a ordonné le traitement des variables.

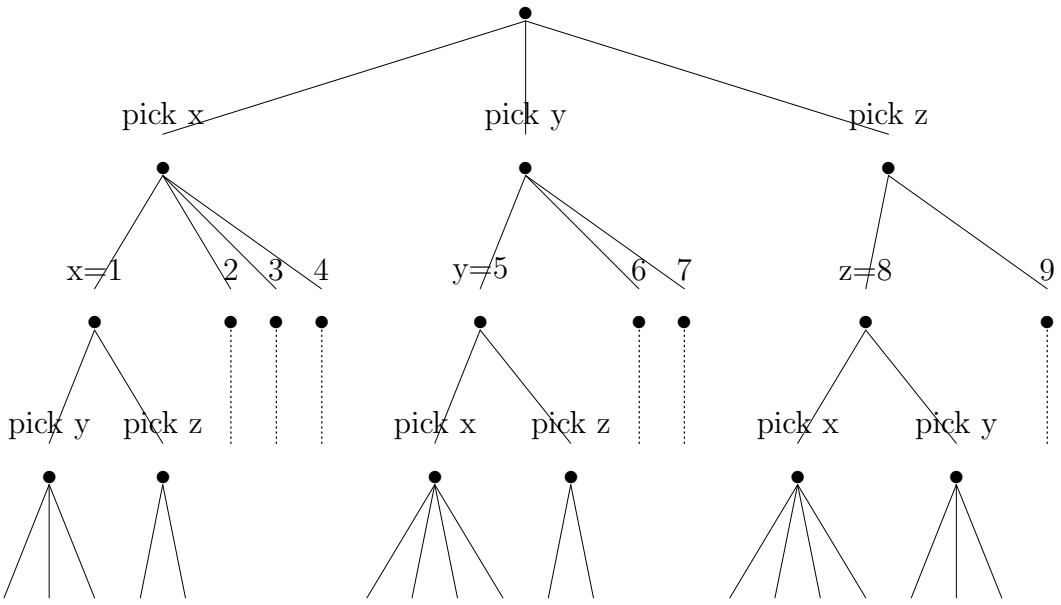


Fig. C.2 : *Espace de recherche d'un CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) où les variables de $\mathcal{X}$* **ne sont pas ordonnées** *et $\mathcal{X} = \{x, y, z\}$, $D_x = \{1, 2, 3, 4\}$, $D_y = \{5, 6, 7\}$ et $D_z = \{8, 9\}$*

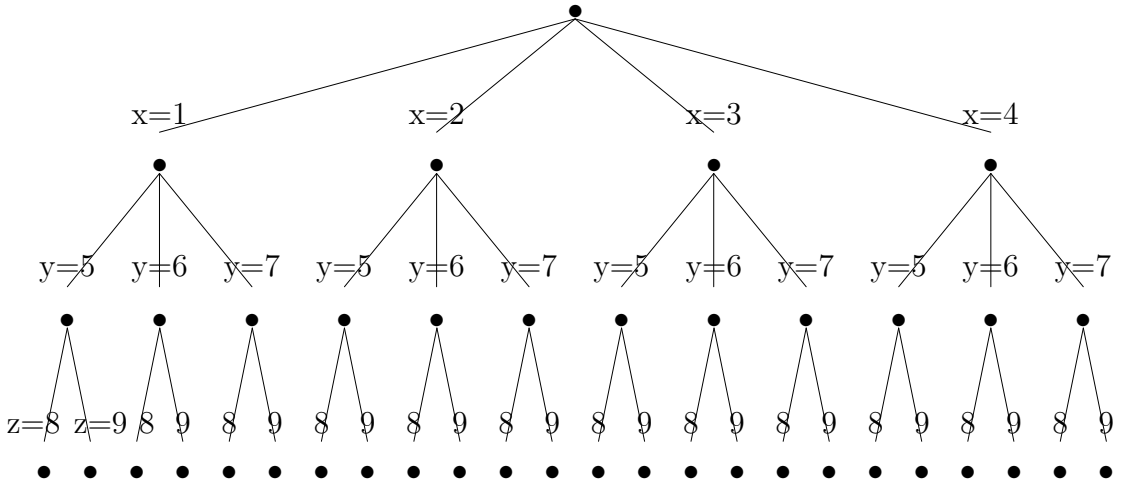


Fig. C.3 : *Arbre de recherche d'un CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) où les variables de $\mathcal{X}$ sont* **ordonnées** *de la façon suivante x, y, z et $\mathcal{X} = \{x, y, z\}$, $D_x = \{1, 2, 3, 4\}$, $D_y = \{5, 6, 7\}$ et $D_z = \{8, 9\}$*

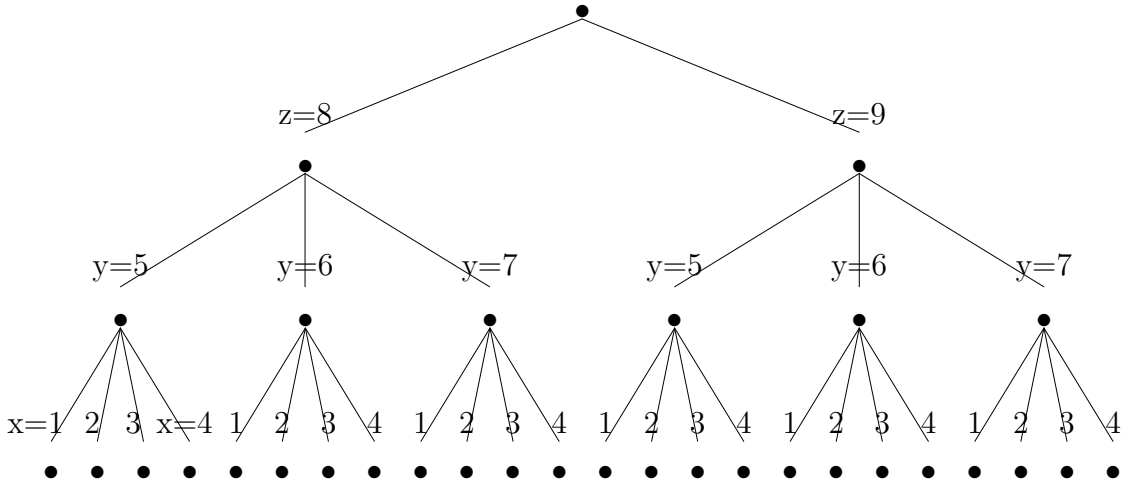


Fig. C.4 : *Arbre de recherche d'un CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) où les variables de $\mathcal{X}$ sont* **ordonnées** *de la façon suivante z, y, x et $\mathcal{X} = \{x, y, z\}$, $D_x = \{1, 2, 3, 4\}$, $D_y = \{5, 6, 7\}$ et $D_z = \{8, 9\}$*

Les figures C.2, C.3 et C.4 montrent l'influence pour un même CSP de la présence ou non d'un ordre sur les variables, et de l'ordre lui-même, sur l'arbre de recherche. La profondeur de l'arbre de recherche et le nombre de feuilles reste respectivement de $2|\mathcal{X}|$ et de $\prod_{\forall x \in \mathcal{X}} |D_x|$. Par contre le nombre de noeuds internes dépend de l'ordre pris sur les variables. Ainsi, si les variables sont ordonnées de celle qui a le plus petit domaine à celle qui a le plus grand domaine, alors le nombre de noeuds internes de l'arbre de recherche sera minimal. De plus, si les variables sont ordonnées, les sous-arbres de chaque branche d'un noeud du même niveau sont identique du point de vue de leur topologie : ce qui est intéressant pour des algorithme de recherche avec un mode d'apprentissage.

C.2.2.3 Choisir une variable

Comme nous l'avons vu, l'ordre de traitement des variables influence largement la taille de l'arbre de recherche. Il existe quatre principales heuristiques pour ordonner le plus efficacement possible les variables d'un problème [116] :

- la stratégie *minimal width ordering* (MWO) ou ordonnancement selon la largeur minimale qui consiste à ordonner les variables en fonction du nombre de variables dont elles dépendent : si une variable x apparaît dans une contrainte c_S alors elle dépend des variables $S \backslash \{x\}$. Celles qui dépendent du plus de variables seront traitées en premier. Dans l'exemple de cryptarithmétique, l'ordre de traitement des variables sera :
$$\left\{ \begin{array}{l} E \\ N \\ O \end{array} \right. > \left\{ \begin{array}{l} S \\ M \\ R \end{array} \right. > \left\{ \begin{array}{l} D \\ Y \end{array} \right. > \left\{ \begin{array}{l} r_2 \\ r_3 \end{array} \right. > r_1 > r_4$$
où $a > b$ signifie que a sera traitée avant b.
- la stratégie *minimal bandwith ordering* (MBO) ou ordonnancement selon la cardinalité minimale qui consiste à choisir au départ n'importe quelle variable puis de choisir ensuite successivement la variable qui partage le plus de contraintes avec les variables déjà choisies. Dans l'exemple de cryptarithmétique, un ordre de traitement des variables à partir de r_4 peut être :
$$r_4 > M > S > O > r_3 > E > N > r_2 > R > r_1 > D > Y$$
- la stratégie *maximum cardinality ordering* (MCO) ou ordonnancement selon le degré maximal qui consiste à choisir au départ n'importe quelle variable qui sera la dernière de l'ordre construit puis de choisir ensuite successivement la variable qui apparaît dans le plus grand nombre de contraintes parmi celles qui portent sur les variables déjà choisies et de la placer avant toutes les variables déjà ordonnées. Dans l'exemple de cryptarithmétique, un ordre de traitement des variables à partir de r_4 peut être :
$$D > Y > r_1 > R > N > E > r_3 > O > S > M > r_4$$

– la stratégie *fail first principe* (FFP) ou principe de l"'échec d'abord" qui consiste à choisir la variable parmi les variables non instanciées qui ont le plus petit domaine.

Les heuristiques MWO, MBO et MCO sont des ordonnancements *statiques* car ils sont opéré avant que la procédure de recherche soit lancée, contrairement à l'heuristique FFP qui est un ordonnancement dynamique puis qu'effectuer à chaque fois qu'une variable doit être choisie pour faire progresser la recherche.

C.2.2.4 Choisir une valeur

Une fois que la variable est choisie, il faut lui affecter une valeur de son domaine "courant". Les heuristiques basée sur l'ordonnancement des valeurs peuvent permettre de trouver la première solution de manière plus efficace si les branches qui ont une meilleure chance d'arriver à la solution peuvent être identifier et essayer en premier.

Cependant, sans utiliser les algorithme d'apprentissage, l'ordonnancement des valeurs ne permet pas d'élaguer un arbre de recherche.

C.2.3 Combiner simplification de problème et *backtracking* simple

L'efficacité de la recherche de solution avec *backtracking* peut être améliorée si l'on peut élaguer les branches de l'arbre de recherche qui n'ont pas de solution. C'est précisément sur ce point que la simplification de problème peut aider : la réduction de la taille du domaine d'une variable revient à élaguer certaines branches, quant au renforcement des contraintes, il permet de réduire l'arbre de recherche dans la dernière phase de recherche. La simplification du problème peut être utilisé à n'importe quel moment lors de la recherche de solution. Il existe de nombreuses stratégies qui combinent la simplification du problème et la recherche de solution comme par exemple les algorithmes[2] de "recherche en avant" *lookahead* [68].

Certaines ont même été prouvées comme étant extrêmement efficace [116].

Exemple 40 (Cryptarithmétique et *backtracking* amélioré)
Reprenons la version minimale du problème "SEND+MORE=MONEY", il s'agit du CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) suivant :

– $\mathcal{X} = \{S, E, N, D, M, O, R, Y, r_1, r_2, r_3, r_4\}$
– $\mathcal{D} = \{D_S, D_E, D_N, D_D, D_M, D_O, D_R, D_Y, D_{r_1}, D_{r_2}, D_{r_3}, D_{r_4}\}$ *avec*
  – $D_E = D_N = D_D = D_R = D_Y = \{2, 3, 4, 5, 6, 7, 8\}$
  – $D_S = \{9\}$

[2]On parle aussi d'algorithmes d"'anticipation".

  - $D_M = D_{r_4} = \{1\}$
  - $D_O = D_{r_3} = \{0\}$
  - $D_{r_1} = D_{r_2} = \{0, 1\}$
- $\mathcal{C} = \{c_1, c_2, c_3, c_4\}$ *avec*
  - $c_1 :\ D + E = Y + 10 \times r_1$
  - $c_2 :\ r_1 + N + R = E + 10 \times r_2$
  - $c_3 :\ r_2 + E = N$
  - $c_4 :\ toutes\_differentes(S, E, N, D, M, O, R, Y)$

Les variables ayant le plus petit domaine sont r_1 *et* r_2. r_1 *est contrainte par 6 variables qui sont* $\{D, E, Y, N, R, r_2\}$ *et* r_2 *par 4 qui sont* $\{r_1, N, R, E\}$. *On va donc commencer par instancier* r_1

- *On instancie la variable* r_1 *à 0.*
  La simplification du problème, nous permet de réduire les domaines de la manière suivante :

$$\begin{cases} D_E = D_D = \{2, 3, 4, 5, 6\} \\ D_N = \{3, 4, 5, 6, 7\} \\ D_R = \{2, 3, 4, 5, 6, 7, 8\} \\ D_Y = \{4, 5, 6, 7, 8\} \\ D_{r_2} = \{0, 1\} \end{cases}$$

  La variable r_2 *possède le plus petit domaine, c'est donc la prochaine variable à instancier.*
  - *On instancie la variable* r_2 *à 0.*
    En utilisant c_3, *on réduit les domaines de* E *et* N *à* $D_E = D_N = \{3, 4, 5, 6\}$.
    En utilisant c_2, *on réduit les domaines de* N, R *et* E *à* $D_N = \{3\}$, $D_R = \{2, 3\}$ *et* $D_E = \{5, 6\}$.
    En réutilisant la contrainte c_3, *il y a une inconsistance.*
    Il faut donc désigner une nouvelle valeur pour r_2.
  - *On instancie la variable* r_2 *à 1.*
    En utilisant c_2, *on réduit les domaines de* N, R *et* E *à* $D_N = \{4, 5, 6, 7\}$, $D_R = \{5, 6, 7, 8\}$ *et* $D_E = \{2, 3, 4, 5\}$.
    En utilisant c_3, *on réduit les domaines de* N *et* E *à* $D_N = \{4, 5, 6\}$ *et* $D_E = \{3, 4, 5\}$.
    En utilisant c_2 *à nouveau, on réduit les domaines de* N, R *et* E *à* $D_N = \{5, 6\}$, $D_R = \{7, 8\}$ *et* $D_E = \{3, 4\}$.
    En utilisant c_3 *à nouveau, on réduit les domaines de* N *et* E *à* $D_N = \{5\}$,*et* $D_E = \{4\}$.
    En réutilisant c_2 *à nouveau, on tombe sur une inconsistance.*
    Il faut donc désigner une nouvelle valeur pour r_2.
    Mais comme cela n'est pas possible, il va falloir remettre en question

l'affectation de la variable r_1.

- *On instancie la variable r_1 à 1.*
  La simplification du problème, nous permet de réduire les domaines de la manière suivante :

$$\begin{cases} D_E = D_N = D_D = \{4,5,6,7,8\} \\ D_R = \{2,3,4,5,6,7,8\} \\ D_Y = \{2,3,4,5,6\} \\ D_{r_2} = \{0,1\} \end{cases}$$

  La variable r_2 possède le plus petit domaine, c'est donc la prochaine variable à instancier.
  - *On instancie la variable r_2 à 0.*
    En utilisant c_2, on réduit les domaines de N, R et E à $D_N = \{4,5\}$, $D_R = \{2,3\}$ et $D_E = \{7,8\}$.
    En utilisant la contrainte c_3, il y a une inconsistance.
    Il faut donc désigner une nouvelle valeur pour r_2.
  - *On instancie la variable r_2 à 1.*
    La simplification du problème, nous permet de réduire les domaines de la manière suivante :

$$\begin{cases} D_E = \{4,5,6,7\} \\ D_D = D_N = D_R = \{5,6,7,8\} \\ D_Y = \{2,3,4,5\} \end{cases}$$

    Les cinq dernières variables ont des domaines de taille identique, mais la variable la plus contrainte est E car elle dépend de toutes les variables, c'est donc elle que nous allons maintenant instancier.
    - *On instancie la variable E à 4.*
      En utilisant la contrainte c_3, on réduit le domaine de N à $D_N = \{5\}$.
      En utilisant la contrainte c_2, on réduit le domaine de R à $D_R = \{8\}$.
      En utilisant la contrainte c_4, on réduit les domaines de D et Y respectivement à $D_D = \{6,7\}$ et $D_Y = \{2,3\}$.
      En utilisant la contrainte c_1, on arrive à une inconsistance.
      Il faut donc désigner une nouvelle valeur pour E.
    - *On instancie la variable E à 5.*
      En utilisant le procédé de réduction du problème, on arrive immédiatement à la solution. En effet, avec la contrainte c_3, on réduit le domaine de N à $D_N = \{6\}$, puis avec la contrainte c_2, on réduit le domaine de R à $D_R = \{8\}$, ensuite, la contrainte c_4 permet de retirer les valeurs 5, 6 et 8 du domaine de D le réduisant à $D_D = \{7\}$ et enfin, en utilisant la contrainte c_1 on réduit le dernier domaine à $D_Y = \{2\}$

Le CSP "SEND+MORE=MONEY" a une solution unique qui est :

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & \\ & & 9 & 5 & 6 & 7 \\ + & & 1 & 0 & 8 & 5 \\ \hline & 1 & 0 & 6 & 5 & 2 \end{array}$$

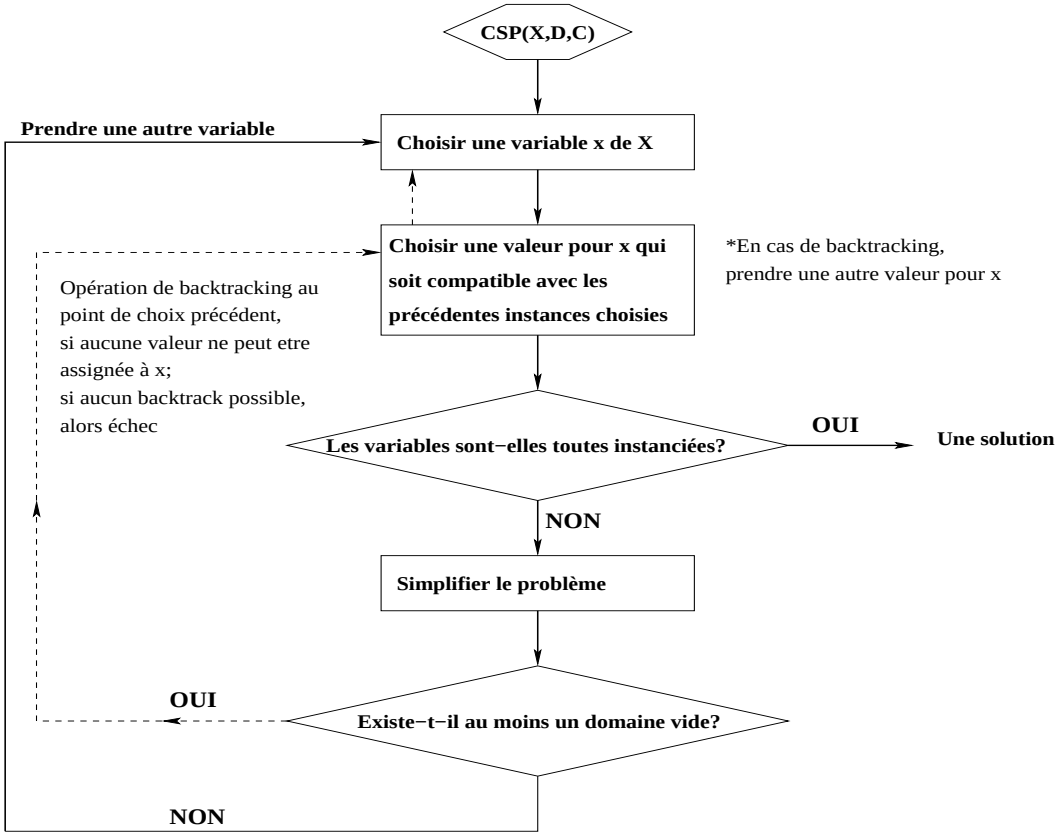


Fig. C.5 : *Schéma général de la stratégie lookahead*

C.3 Synthétiser les solutions (*Solution synthesis*)

La synthèse de solution a été introduite pour la première fois par Freuder [57] : elle peut être vue à la fois comme un algorithme de recherche qui explore plusieurs branches simultanément ou comme une simplification du problème dans lequel la contrainte pour toutes les variables (i.e. la contrainte à n variables d'un problème à n variables) est créée et réduite à un ensemble qui contient toutes les solutions et uniquement celles-la. Ce qui distingue cette approche est que la solution est générer de manière constructive. Cette approche est très utilisée pour des problèmes extrêmement contraints et pour lesquels on aimerait obtenir toutes les solutions.

Lors de la recherche, une affectation partielle est regardée à la fois. Cette affectation partielle est étendue en rajoutant une instance à chaque étape jusqu'à ce qu'une solution soit trouvée ou que toutes les affectations partielles aient été essayées. L'idée de base d'une synthèse de solution est de collecter les affectations partielles possibles en agrandissant petit à petit l'ensemble des variables considérées jusqu'à le faire pour l'ensemble de toutes les variables. La figure C.6 résume schématiquement cette technique.

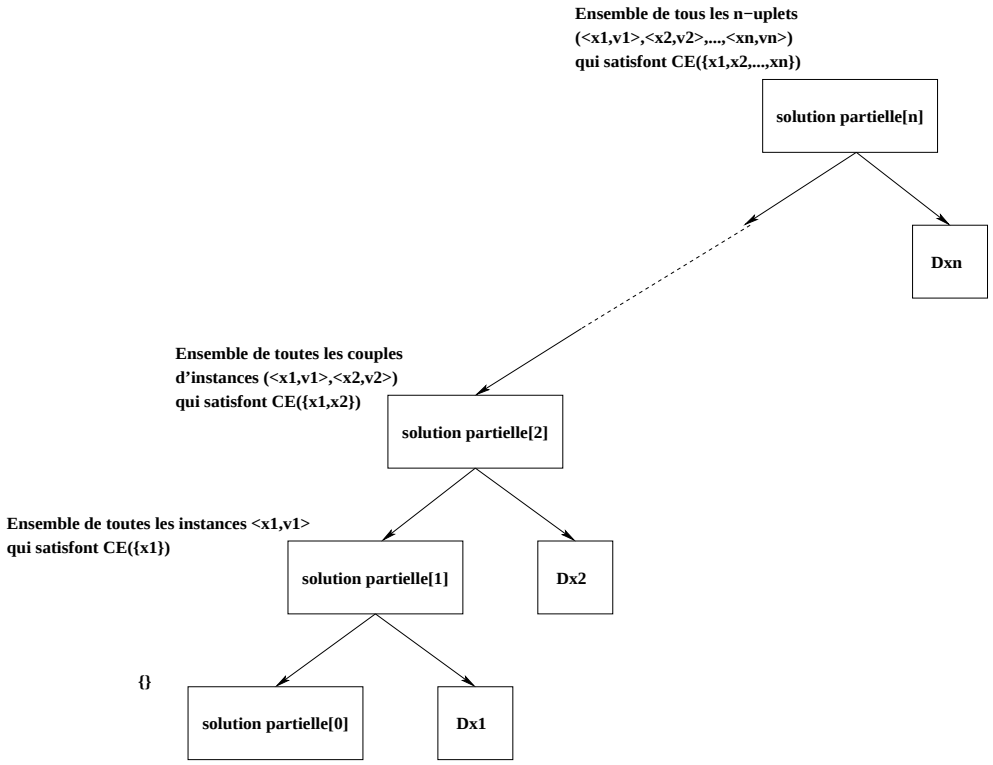


Fig. C.6 : *Schéma général de l'algorithme de synthèse de solutions*

Définition 22 (expression contrainte) *Une expression contrainte sur un ensemble de variables S est une collection de contraintes sur S et sur ses sous-ensembles. On note $CE(S)$ une expression contrainte.*

Définition 23 *Une affectation A satisfait une expression contrainte CE si A satisfait toutes les contraintes de CE.*

Exemple 41 (Cryptarithmétique et synthèse de solutions)
Reprenons le problème de départ du "SEND+MORE=MONEY', soit le CSP($\mathcal{X}, \mathcal{D}, \mathcal{C}$) suivant :

- $\mathcal{X} = \{S, E, N, D, M, O, R, Y, r_1, r_2, r_3, r_4\}$
- $\mathcal{D} = \{D_S, D_E, D_N, D_D, D_M, D_O, D_R, D_Y, D_{r_1}, D_{r_2}, D_{r_3}, D_{r_4}\}$ *avec*
  - $D_S = D_E = D_N = D_D = D_M = D_O = D_R = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - $D_{r_1} = D_{r_2} = D_{r_3} = D_{r_4} = \{0, 1\}$
- $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ *avec*
  - $c_1 :\ D + E = Y + 10 \times r_1$
  - $c_2 :\ r_1 + N + R = E + 10 \times r_2$
  - $c_3 :\ r_2 + E + O = N + 10 \times r_3$
  - $c_4 :\ r_3 + S + M = O + 10 \times r_4$
  - $c_5 :\ r_4 = M$
  - $c_6 :\ M \neq 0$
  - $c_7 :\ S \neq 0$
  - $c_8 :\ toutes\_differentes(S, E, N, D, M, O, R, Y)$

On part de l'affectation partielle $I_0 = \emptyset$.

étape 1 *On choisit la variable M car c'est celle sur laquelle porte le plus de contraintes.*
$CE(M) = \{c_4, c_5, c_6, c_8\}$ et $D_M = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
La contrainte c_5 permet de réduire le domaine des valeurs possibles pour M à l'ensemble $\{0, 1\}$ et la contrainte c_6 oblige à retirer de cet ensemble la valeur 0. Par conséquent, seules les affectations contenant l'instance $(M, 1)$ sont candidates.

$$I_1 = \{\{(M, 1)\}\}$$

étape 2 *On choisit la variable r_4 par conséquence de l'étape 1 et de la contrainte c_5.*
$CE(\{M, r_4\}) = CE(\{M\})$ et $D_{r_4} = \{0, 1\}$.
À partir de la contrainte c_5 et de l'affectation partielle construite à cette étape, on en déduit que l'instance $(r_4, 1)$ fait partie de toute affectation solution.

$$I_2 = \{\{(M, 1), (r_4, 1)\}\}$$

étape 3 *On choisit la variable r_3 pour la taille de son domaine et qu'une des deux contraintes qui la concerne est déjà dans le CE.*
$CE(\{M, r_4, r_3\}) = CE(\{M, r_4\}) \bigcup \{c_3\}$ *et* $D_{r_3} = \{0, 1\}$.
On a alors deux affectations partielles candidates :

$$I_3 = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,0)\} \\ \{(M,1),(r_4,1),(r_3,1)\} \end{array} \right\}$$

étape 4 *On choisit la variable r_2 pour la taille de son domaine et qu'une des deux contraintes qui la concerne est déjà dans le CE.*
$CE(\{M, r_4, r_3, r_2\}) = CE(\{M, r_4, r_3\}) \bigcup \{c_2\}$ *et* $D_{r_2} = \{0, 1\}$.
On a alors quatre affectations partielles candidates :

$$I_4 = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,0),(r_2,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,1)\} \end{array} \right\}$$

étape 5 *On choisit la variable r_1 pour la taille de son domaine et qu'une des deux contraintes qui la concerne est déjà dans le CE.*
$CE(\{M, r_4, r_3, r_2, r_1\}) = CE(\{M, r_4, r_3, r_2\}) \bigcup \{c_1\}$ *et* $D_{r_1} = \{0, 1\}$.
On a alors huit affectations partielles candidates :

$$I_5 = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,0),(r_2,0),(r_1,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,0),(r_1,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,1),(r_1,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,0),(r_1,1)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,0),(r_1,1)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,1),(r_1,1)\} \end{array} \right\}$$

étape 6 *On choisit la variable S parce que toutes les contraintes qui la concernent sauf une sont déjà dans le CE.*
$CE(\{M, r_4, r_3, r_2, r_1, S\}) = CE(\{M, r_4, r_3, r_2, r_1\}) \bigcup \{c_7\}$ *et*
$D_S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
La contrainte c_7 permet de retirer la valeur 0 du domaine des valeurs possibles pour S et la contrainte c_8 couplée au fait que toute affectation solution contient l'instance $(M, 1)$ permet de retirer la valeur 1.

Considérons la contrainte c_4 : $r_3 + S + M = O + 10 \times r_4$ qui se réduit en $r_3 + S = O + 9$, comme $O \geq 0$ (par son domaine) alors $O + 9 \geq 9$ donc $r_3 + S \geq 9$. r_3 pouvant prendre les valeurs 0 ou 1, on en déduit que $S \geq 9$

ou $S \geq 8$, ce qui nous donne $S \geq 8$. On peut alors réduire le domaine des valeurs possibles pour S à un ensemble de deux valeurs $\{8, 9\}$.

Considérons toujours la contrainte c_4 mais avec les affectations partielles suivantes :

- *$\{(r_3, 0), (S, 8)\}$ alors la contrainte c_4 se réduit en $8 = O + 9$. Comme la variable O ne peut pas être négative, on en déduit que toute affectation contenant l'affectation partielle $\{(r_3, 0), (S, 8)\}$ n'est pas solution*
- *$\{(r_3, 1), (S, 9)\}$ alors la contrainte c_4 se réduit en $10 = O + 9$ mais pour la satisfaire, il faut que $O = 1$ ce qui n'est pas possible à cause de la contrainte c_8 et du fait que toute solution contient l'instance $(M, 1)$. On en déduit que toute affectation contenant l'affectation partielle $\{(r_3, 1), (S, 9)\}$ n'est pas solution.*
- *$\{(r_3, 0), (S, 9)\}$ alors la contrainte c_4 se réduit en $9 = O + 9$ ce qui à première vue est satisfiable avec $O = 0$ sans violer directement d'autres contraintes.*
- *$\{(r_3, 1), (S, 8)\}$ alors la contrainte c_4 se réduit en $9 = O + 9$ ce qui à première vue est satisfiable avec $O = 0$ sans violer directement d'autres contraintes.*

On a alors huit affectations partielles candidates :

$$I_6 = \left\{ \begin{array}{l} \{(M, 1), (r_4, 1), (r_3, 1), (r_2, 0), (r_1, 0), (S, 8)\} \\ \{(M, 1), (r_4, 1), (r_3, 1), (r_2, 1), (r_1, 0), (S, 8)\} \\ \{(M, 1), (r_4, 1), (r_3, 1), (r_2, 0), (r_1, 1), (S, 8)\} \\ \{(M, 1), (r_4, 1), (r_3, 1), (r_2, 1), (r_1, 1), (S, 8)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 0), (r_1, 0), (S, 9)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 0), (r_1, 1), (S, 9)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9)\} \end{array} \right\}$$

étape 7 *On choisit la variable O qui est la seule variable de la contrainte c_4 non traitée.*

$CE(\{M, r_4, r_3, r_2, r_1, S, O\}) = CE(\{M, r_4, r_3, r_2, r_1, S\})$ *et*
$D_O = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

La contrainte c_8 couplée au fait que toute affectation solution contient l'instance $(M, 1)$ permet de retirer la valeur 1 du domaine des valeurs possibles de O.

Considérons la contrainte c_4 réduite $r_3 + S = O + 9$ en mais avec les affectations partielles suivantes :

- *$\{(r_3, 0), (S, 9)\}$ alors la contrainte c_4 se réduit en $9 = O + 9$ qui est satisfaite avec $O = 0$.*
- *$\{(r_3, 1), (S, 8)\}$ alors la contrainte c_4 se réduit en $9 = O + 9$ qui est satisfaite avec $O = 0$.*

On en déduite que toute affectation solution contient l'instance $(O, 0)$. *On a alors toujours huit affectations partielles candidates :*

$$I_7 = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,1),(r_2,0),(r_1,0),(S,8),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,1),(r_1,0),(S,8),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,0),(r_1,1),(S,8),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,1),(r_2,1),(r_1,1),(S,8),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,0),(r_1,0),(S,9),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,0),(S,9),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,0),(r_1,1),(S,9),(O,0)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0)\} \end{array} \right\}$$

étape 8 *On choisit la variable* E *qui est la variable non traitée sur laquelle porte le plus de contraintes.*
$CE(\{M, r_4, r_3, r_2, r_1, S, O, E\}) = CE(\{M, r_4, r_3, r_2, r_1, S, O\})$ *et*
$D_E = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
La contrainte c_8 *couplée au fait que toute affectation solution contient les instances* $(M, 1)$ *et* $(O, 0)$ *permet de retirer les valeurs 0 et 1 du domaine des valeurs possibles de* E.

Considérons la contrainte c_3 *qui se réduite en* $r_2 + E = N + 10 \times r_3$ *et les affectations partielles suivantes :*

- $\{(r_3, 0), (r_2, 0)\}$ *alors la contrainte* c_3 *se réduit en* $E = N$ *ce qui viole la contrainte* c_8. *L'affectation partielle* $\{(r_3, 0), (r_2, 0)\}$ *ne fait partie d'aucune affectation solution.*
- $\{(r_3, 1), (r_2, 0)\}$ *alors la contrainte* c_3 *se réduit en* $E = N + 10$ *qui est insatisfiable car* $2 \leq E \leq 9$ *et la variable* N *ne peut pas avoir de valeur négative. L'affectation partielle* $\{(r_3, 1), (r_2, 0)\}$ *ne fait partie d'aucune affectation solution.*
- $\{(r_3, 1), (r_2, 1)\}$ *alors la contrainte* c_3 *se réduit en* $E + 1 = N + 10$ *qui est insatisfiable car*

$$\begin{array}{ll} & 2 \leq E \leq 9 \\ \Leftrightarrow & 3 \leq E + 1 \leq 10 \\ \Leftrightarrow & 3 \leq N + 10 \leq 10 \text{ par la contrainte } c_3 \\ \Leftrightarrow & -13 \leq N \leq 0 \end{array}$$

Or d'après le domaine de N, *la seule valeur possible est* 0 *mais cela viole la contrainte* c_8 *car l'instance* $(O, 0)$ *fait partie de toute affectation solution. L'affectation partielle* $\{(r_3, 1), (r_2, 1)\}$ *ne fait partie d'aucune affectation solution.*

- $\{(r_3, 0), (r_2, 1)\}$ *alors la contrainte* c_3 *se réduit en* $E + 1 = N$ *ce qui est satisfiable sous la condition que* E *ne puisse pas prendre la valeur 9.*

On en déduit que toute affectation solution contient les instances $(r_3, 0)$ *et* $(r_2, 1)$ *et que le domaine de la variable* E *se réduit à* $\{2, 3, 4, 5, 6, 7, 8\}$.

L'ensemble des affectations candidats de l'étape précédent se réduit alors à deux éléments $\{\{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0)\}$ *et* $\{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0)\}\}$. *On peut remarquer qu'à cette étape, on est sûr que toute solution contient l'instance* $(S, 9)$. *En utilisant cette information, la contraint* c_8 *et la contrainte* c_3 *réduite à* $E + 1 = N$, *on peut encore retirer la valeur 8 du domaine des valeurs de* E.

On a alors douze affectations partielles candidates :

$$I_8 = \left\{ \begin{array}{l} \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 2)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 2)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 3)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 3)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 4)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 4)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 5)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 5)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 6)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 6)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 0), (S, 9), (O, 0), (E, 7)\} \\ \{(M, 1), (r_4, 1), (r_3, 0), (r_2, 1), (r_1, 1), (S, 9), (O, 0), (E, 7)\} \end{array} \right\}$$

étape 9 *On choisit la variable* R.
$CE(\{M, r_4, r_3, r_2, r_1, S, O, E, R\}) = CE(\{M, r_4, r_3, r_2, r_1, S, O, E\})$ *et* $D_R = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
La contrainte c_8 *couplée au fait que toute affectation solution contient les instances* $(M, 1)$, $(O, 0)$ *et* $(S, 9)$ *permet de retirer les valeurs 0, 1 et 9 du domaine des valeurs possibles de* R.

Considérons la contrainte $c_2 : \ r_1 + N + R = E + 10 \times r_2$ *qui se réduit en* $r_1 + N + R = E + 10$ *et la contrainte* c_3 *réduite en* $E + 1 = N$, *on a alors* $r_1 + E + 1 + R = E + 10$ *qui se réduit en* $r_1 + R = 9$. *Par la contrainte* c_8 *et le fait que toute solution contient* $(S, 9)$, *on en déduit que la variable* r_1 *ne peut pas être de valeur nulle. Par conséquence, les instances* $(r_1, 1)$ *et* $(R, 8)$ *font partie de toute affectation solution.*

En utilisant c_8, c_3 *réduit à* $E+1 = N$ *et la présence de l'instance* $(R, 8)$ *dans toute solution, on en déduit que la valeur 7 n'est plus une valeur permise pour* E.

On a alors cinq affectations partielles candidates :

$$I_9 = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,2),(R,8)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,3),(R,8)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,4),(R,8)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,5),(R,8)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,6),(R,8)\} \end{array} \right\}$$

étape 10 *On choisit la variable N.*

$CE(\{M,r_4,r_3,r_2,r_1,S,O,E,R,N\}) = CE(\{M,r_4,r_3,r_2,r_1,S,O,E,R\})$ *et* $D_N = \{0,1,2,3,4,5,6,7,8,9\}$.

La contrainte c_8 couplée au fait que toute affectation solution contient les instances $(M,1)$, $(O,0)$, $(S,9)$ et $(R,8)$ permet de retirer les valeurs 0, 1, 8 et 9 du domaine des valeurs possibles de N.

En intégrant la contrainte c_3 réduite à $E+1=N$ dans les affectations candidates, on obtient alors :

$$I_{10} = \left\{ \begin{array}{l} \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,2),(R,8),(N,3)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,3),(R,8),(N,4)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,4),(R,8),(N,5)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,5),(R,8),(N,6)\} \\ \{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,6),(R,8),(N,7)\} \end{array} \right\}$$

étape 11 *On choisit la variable D.*

$CE(\{M,r_4,r_3,r_2,r_1,S,O,E,R,N,D\}) = CE(\{M,r_4,r_3,r_2,r_1,S,O,E,R,N\})$ *et* $D_D = \{0,1,2,3,4,5,6,7,8,9\}$.

La contrainte c_8 couplée au fait que toute affectation solution contient les instances $(M,1)$, $(O,0)$, $(S,9)$ et $(R,8)$ permet de retirer les valeurs 0, 1, 8 et 9 du domaine des valeurs possibles de D.

Considérons la contraintes c_1 et les différentes affectations candidates :

- $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,2),(R,8),(N,3)\}$ *permet de réduire la contrainte c_1 à $D+2=Y+10$ soit $D=Y+8$ qui n'est pas satisfiable car les seules couples d'instances, i.e. ($(D,8)$,$Y,0$) et ($(D,9)$,$(D,1)$), qui la satisfassent violent la contrainte c_8. Cette affectation partielle ne peut pas faire partie d'une affectation solution.*
- $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,3),(R,8),(N,4)\}$ *permet de réduire la contrainte c_1 à $D=Y+7$ qui n'est pas satisfiable car les seules couples d'instances qui la satisfassent violent la contrainte c_8. Cette affectation partielle ne peut pas faire partie d'une affectation solution.*
- $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,4),(R,8),(N,5)\}$ *permet de réduire la contrainte c_1 à $D=Y+6$ qui n'est pas satisfiable*

car les seules couples d'instances qui la satisfassent violent la contrainte c_8. Cette affectation partielle ne peut pas faire partie d'une affectation solution.

- $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,5),(R,8),(N,6)\}$ *permet de réduire la contrainte c_1 à $D = Y + 5$ qui est satisfiable sans violer la contrainte c_8.*
- $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,6),(R,8),(N,7)\}$ *permet de réduire la contrainte c_1 à $D = Y + 4$ qui n'est pas satisfiable car les seules couples d'instances qui la satisfassent violent la contrainte c_8. Cette affectation partielle ne peut pas faire partie d'une affectation solution.*

L'ensemble des affectations partielles candidates se réduit à l'affectation partielle $\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,5),(R,8),(N,6)\}$ *que l'on complète à l'aide de la contrainte c_1 par le seul couple d'instances possible qui est (*$(D,7)$,$(Y,2)$*).*

La seule solution de ce CSP est

$$\{(M,1),(r_4,1),(r_3,0),(r_2,1),(r_1,1),(S,9),(O,0),(E,5),(R,8),(N,6),(D,7),(Y,2)\}$$

Bibliographie

[1] Preciflash du 15 octobre 2002.
http ://www.precilog.com/preciflash_15_10_2002_fr.php.

[2] The economic impacts of inadequate infrastructure for software testing. Technical report, RTI, 2002. RTI Project Number 7007.011
http ://www.nist.gov/director/prog-ofc/report02-3.pdf.

[3] *Special section on Software Inspection*, volume 29, pages 674–733. IEEE Transactions on Software Engineering, august 2003.

[4] J.-R. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. PhD thesis, Université de Rennes I, IRISA, 1998. No 1869.

[6] L. Van Aertryck, M.V. Benveniste, and D. Le Métayer. CASTING : a formally based software test generation method. In *The 1st International Conference on Formal Engineering Methods (ICFEM'97), IEEE*, pages 101–110, 1997.

[7] L. Van Aertryck and T. Jensen. UML-CASTING : Test synthesis from UML models using constraint resolution. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2003.

[8] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of Principles of Programming Languages (POPL'94)*, pages 25–34, 1994.

[9] A. Aho, R. Sethi, and J. Ullman. *COMPILATEURS : Principes, techniques et outils*. Collection Informatique Intelligence Artificielle. InterEditions, 1989. ISBN 2-7296-0295-X.

[10] S.B. Akers. Binary decision diagrams. In *IEEE Transactions on Computers*, pages 509–516, 1978.

[11] B. Allo. L'automatisation des tâches de validation fonctionnelle et structurelle. Conférence Ingénieurie Automobile, novembre 2003.
http ://axlog.fr/sect/conf_automobile/Allo.pdf.

[12] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools : A tool-set for test generation from Z and B using constraint logic programming. In *FATES'02, Formal Approaches to Testing of Software, Workshop of CONCUR'02*, pages 105–120, Brnö, Czech Republic, August 2002.

[13] A. Arnould. *Test à partir de spécifications de structures bornées : une théorie du test, une méthode de sélection , un outil d'assistance à la sélection*. PhD thesis, Université Paris-Sud-Orsay, LRI, 1997. No 4689.

[14] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications : GSM 11.11 standard case-study. *Software Pratice and Experience*, 2002. Rapport de fin de contrat, à paraître dans International Journal on Software-Practice and Experience.

[15] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405, novembre 1991.

[16] A. Bertolino and M. Marré. Automatic generation of paths covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12) :885–899, décembre 1994.

[17] A. Bertolino, R. Mirandola, and E. Peciola. A case study in branch testing automation. *The Journal of Systems and Software*, 38(1) :47–59, 1997.

[18] M. Bidoit and P.D. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language*. LNCS 2900. Springer-Verlag, 2004.

[19] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti. Pods-a project on diverse software. In *IEEE Transactions on Software Engineering*, volume SE-12, pages 929–941, 1986.

[20] G. Booch, I. Jacobson, and J. Rumbaugh. The Unified Modeling Language for object-oriented development version 1.0, 1997. Rational Software Corporation.

[21] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B : A constraint solver for B. In *Proceedings of the COnference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, 2002.

[22] L. Du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre*. PhD thesis, Université Joseph Fourier - Grenoble I, septembre 1999.

[23] L. Du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance testing from UML specifications. In A. Evans, R.B. France, A.M.D. Moreira, and B. Rumpe, editors, *Pratical UML-Based Rigorous Development Methods-Counter or Integrating the eXtremists, Workshop of the pUML-Group with the UML2001*, pages 43–55, 2001.

[24] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation : a synchronous point of view. In *Feature Interactions in Telecommunications Systems V*. K. Kimble and L.G. Bouma editors, IOS Press, 1998.

[25] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess : a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering, ACM*, may 1999.

[26] L. Du Bousquet and N. Zuanon. An overview of Lutess : A specification-based tool for testing synchronous software. In *14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 208–215, octobre 1999.

[27] L. Briand and Y. Labiche. A UML-based approach to system testing. In *Software and Systems Modeling*, volume 1, pages 10–42, 2002.

[28] N. Caritey, L. Gaspari, B. Legeard, and F. Peureux. Specification-based testing - application on algorithms of Metro and RER tickets (confidential). Technical report, LIFC, University of Franche-Comté and Schlumberger Besanon, 2001. TR-03/01.

[29] T.S. Chow. Testing software design modeled by finite-state machines. In *IEEE Transaction on Software Engineering*, volume 4, pages 178–187, 1978.

[30] H. Chu, J. Dobson, and I.Liu. FAST : A framework for automating statistics-based testing. In *Software Quality Journal*, pages 13–36, 1997.

[31] Ouvrage collectif. Opération 2 : vérification et génération de tests pour un système de comptage de neutrons. Rapport annuel et actes du colloque FORMA, janvier 1998.

[32] S. Corteel. *Problème énumératifs issus de l'Informatique, de la Physique et de la Combinatoire*. PhD thesis, Université Paris-Sud-Orsay, janvier 2000.

[33] S. Corteel, A. Denise, I. Dutour, F. Sarron, and P. Zimmermann. CS web page. http ://dept-info.labri.u-bordeaux.fr/∼dutour/CS/.

[34] Y. Crouzet, P. Thévenod-Fosse, and H. Waeselynck. Validation du test du logiciel par injection de fautes : l'outil SESAME. In *11ème Colloque National de Fiabilité & Maintenabilité*, pages 551–559, 1998. Rapport LAAS No98249.

[35] R. Cytron, J. Ferrante, B.K Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *Transactions on Programming Languages and Systems*, volume 13, pages 451–490, octobre 1991.

[36] P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing : a case study on the software of an automatic subway.

Journal of Systems and Software, 21(3) :229–244, 1993. Edition spéciale : Applying Specification, Verification & Validation Techniques.

[37] M. Davis. Hilbert's tenth problem is unsolvable. In *American Math Monthly*, pages 233–269, 1973.

[38] R.A. DeMillo. Mutation analysis as a tool for software quality assurance. In *Proceedings COMPSAC'80*, pages 390–393, 1980.

[39] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, 1988.

[40] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection : help for the praticing programmer. *IEEE Computer Magazine*, 11(4) :34–41, avril 1978.

[41] R.A. DeMillo and A.J. Offut. Constraint-based automatic test data generation. *IEEE Transaction on Software Engineering*, 17(9) :900–910, septembre 1991.

[42] A. Denise. Structures aléatoires, modèles et analyse des génomes. Technical Report 1301, Université Paris-Sud-Orsay, LRI, 2002.

[43] A. Denise, I. Dutour, and P. Zimmermann. CS : a package for counting and generating combinatorial structures. *mathPAD*, 8(1) :22–29, 1998. http ://www.mupad.de/mathpad.shtml.

[44] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars, definitions, systems and bibliography.

[45] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *In First International Symposium of Formal Methods Europe (FME'93)*, LNCS 670, pages 268–284. Springer-Verlag, 1993.

[46] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Random sampling from boltzmann principes. In P. Widmayer et al, editor, *ICALP 2002*, LNCS 2380, pages 501–513. Springer-Verlag.

[47] S. Dupuy-Chessa and L. du Bousquet. Validation of UML models thanks to Z and LUSTRE. In J.N. Oliveira and P. Zave, editors, *Formal Methods for Increasing Software Productivity : International Symposium of Formal Methods Europe (FME)*, LNCS 2021, pages 242–258. Springer-Verlag, 2001.

[48] J.W. Duran and S.C. Ntafos. A report on random testing. *5th IEEE International Conference on Software Engineering*, pages 179–183, March 1981.

[49] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10 :438–444, July 1984.

[50] M. Dyer. *The cleanroom approach to quality software development*. John Wiley & Sons, 1992. ISBN 0-471-54823-5.

[51] *Website of the ECL^{i}PS^e Constraint Logic Programming System.* http ://www.icparc.ic.ac.uk/eclipse/.

[52] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linkoping, ECSEL*, pages 21–28, octobre 1999.

[53] Laboratoires VERIMAG et LSR de Grenoble et Schneider Electric. Projet AUTOFOR. http ://www.systemes-critiques.org/autofor_fr.php.

[54] A. Denise et P. Zimmermann. Uniform random generation of decomposable structures using floating-point arithmetic. *Theoretical Computer Science*, (218) :233–248, 1999. Rapport INRIA No3242, septembre 1997.

[55] R. Ferguson and B. Korel. The chaining approach for software test data generation. *IEEE Transactions on Software Engineering*, 5(1) :63–86, janvier 1996.

[56] Ph. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132 :1–35, 1994. RR INRIA No 1830.

[57] E.C. Freuder. Synthesizing constraint expressions. In *Communication of ACM*, volume 21, pages 958–966. novembre 1978.

[58] M.-C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. *Précis de génie logiciel*. Masson, 1996. ISBN 2-225-85189-1.

[59] M. Goldwurm. Random generation of words in an algebraic language in linear binary space. In *Information Processing Letters*, number 54, pages 229–233, 1995.

[60] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In ACM SIGSOFT, editor, *Proc. ISSTA 98 (Symposium on Software Testing and Analysis)*, volume 2, pages 53–62, 1998.

[61] S.-D. Gouraud. Application de la génération aléatoire de structures combinatoires au test de logiciel. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, pages 99–112, 2001.

[62] S.-D. Gouraud. AuGuSTe : un outil pour le test statistique. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, pages 337–340, 2004.

[63] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Automated Software Engineering Conference, IEEE*, pages 5–12, 2001.

[64] N.D. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Otha. Feature interaction detection context. In *Feature Interactions in Telecommunications Systems V*, pages 327–359. K. Kimble and L.G. Bouma editors, IOS Press, 1998.

[65] N. Gupta, A.P. Mathur, and M.L. Soffa. Generating test data for branch coverage. In *15th IEEE International Conference on Automated Software Engineering*, pages 219–227, septembre 2000.

[66] N. Halbwachs, P. Casti, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, septembre 1991.

[67] M.H. Halstead. *Elements of Software Science*. Elsevier Science, 1977.

[68] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[69] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM. J. Comput*, 12(4) :645–655, 1983.

[70] Open systems interconnection International Organisation for Standardisation. Conformance testing methodology and framework, 1992. ISO 9646.

[71] T. Jéron. Le test de conformité : état de l'art. Rapport pour l'AAE (Architecture Electronique Embarquée), 2001.
http ://www.irisa.fr/vertecs/Publis/Ps/2001-AEE_test.ps.

[72] M. Jerrum and A. Sinclair. The markov chain monte-carlo method : an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems*, pages 482–520, 1996. In D.S. Hochbaum, editor, PWS Publishing.

[73] C.B. Jones. Systematic software development using VDM, 1990. Prentice Hall Int.

[74] C. Kaner. Software negligence and testing coverage, 1996.
http ://www.kaner.com/coverage.htm.

[75] N. K. Karmarkar. A new polynomial time algorithm for linear programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

[76] N. K. Karmarkar. Interior-point methods in optimization. In *Proceedings of the Second International Conference on Industrial and Applied Mathematics (ICIAM'91), Washington, DC, July, 1991*, pages 160–181. SIAM Publications, Philadelphia, PA, USA, 1992.

[77] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, and P. Tonella. Automatic unit test data generation using mixed-integer linear programming and execution trees. In *The International Conference on Software Maintenance ICSM*, pages 189–198, 1999.

[78] J.-C. Laprie, B. Courtois, M.-C. Gaudel, and D. Powell. *Sûreté de fonctionnement des systèmes informatiques*. ISBN 2-04-016942-3, 1989.

[79] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, août 1996.

[80] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing grom Z and B. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe*, volume 2391 of LNCS of *Springer-Verlag*, pages 21–40, 2002.

[81] Anders Møller. Finite-state automata for java.
http ://www.brics.dk/ amoeller/automaton/.

[82] D. Lugato, C. Bigot, and Y. Valot. Validation and automatic test generation on UML models : the AGATHA approach. 2002.

[83] N.A. Lynch. I/O automata : A model for discrete event systems. In *Proceedings of 22nd Conference on Information Sciences and Systems*, pages 29–38, Princeton, NJ, USA, mars 1988.

[84] A.K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58(1-3) :3–20, 1992. Special Volume on Constraint Based Reasoning.

[85] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In MIT Press, editor, *ICLP'91, Eight International Conference on Logic Programming*, pages 25–28, 1991.

[86] B. Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. PhD thesis, Université Paris-Sud-Orsay, LRI, 1991.

[87] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *15h IEEE International Conference on Automated Software Engineering*, pages 229–237, 2000.

[88] B. Marre, P. Mouy, and N. Williams. On-the-fly generation of structural tests for C functions. In *16th International Conference Software and Systems Engineering and their applications (ICSSEA'2003)*, 2003.

[89] B. Marre, P. Thévenod-Fosse, H. Waeselynck, P. Le Gall, and Y. Crouzet. An experimental evaluation of formal testing and statistical testing. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 273–281. Springer, 1995. ISBN 3-540-59334-9.

[90] M. Marré and A. Bertolino. Using spanning sets for coverage testing. In *IEEE Transactions on Software Engineering*, volume 29, pages 974–984, novembre 2003.

[91] P. Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. PhD thesis, Université de Rennes I, février 2000. N° 2320.

[92] P.D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. LNCS 2960. Springer-Verlag, 2004.

[93] A. Nijenhuis and H.S. Wilf. *Combinatorial algorithms*. Computer Science and Applied Mathematics. Academic Press, Harcourt Brace Jovanovich Publishers, New York, 1975.

[94] National Institute of Standards and Technology. http ://www.nist.gov.

[95] I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16h IEEE International Conference on Automated Software Engineering*, pages 364–367, 2001.

[96] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.

[97] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J ;-M. Jézéquel, and A. Le Guennec. System test synthesis from UML models od distributed software. In D.A. Peled and M.Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems FORTE : 22nd IFIP WG 6.1 International Conference*, LNCS 2529, pages 97–113. Springer-Verlag, 2002.

[98] R.E. Prather and J.P. Myers. The path prefix software testing strategy. *IEEE Transactions on Software Engineering*, 13(7) :761–766, july 1987.

[99] J.G. Propp. Generating random elements of a finite distribution lattice. *Electronic Journal of Combinatorics*, 4(2), 1997. R15.

[100] J.G. Propp and D.B. Wilson. How to get a perfectly random sample from a generic markov chain and generate a random spanning tree of a directed graph. *Journal of Algorithms*, 27(2) :170–217, mai 1998.

[101] C.V. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4) :293–300, 1976.

[102] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*.

[103] V. Rusu, L. Du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, LNCS 1945, pages 338–357. Springer-Verlag, 2000.

[104] G. Schaeffer. *Conjugaison d'arbres et cartes combinatoires aléatoires*. PhD thesis, Université Bordeaux I, 1998.

[105] G. Schaeffer. Random sampling of large planar maps and convex polyhedra. In *Proceedings of STOC'99*, Atlanta, 1999.

[106] R. Segala. *Quiescence, fairness and the notion of implementation*. In *CONCUR*, LNCS 715. Springer-Verlag, 1993.

[107] C. Solnon. *Programmation par Contraintes*. http ://www710.univ-lyon1.fr/ csolnon/Site-PPC/e-miage-ppc-som.htm.

[108] H. Tardieu, A. Rochfeld, and R. Coletti. *La méthode MERISE, principes et outils*. 1991.

[109] The MuPAD Group (Benno Fuchssteiner et al.). *MuPAD User's Manual - MuPAD Version 1.2.2 Multi Processing Algebra Data Tool*. John Wiley and sons, 1996. http ://www.mupad.de/.

[110] P. Thévenod-Fosse. Software validation by means of statistical testing : Retrospect and future direction. In *International Working Conference on Dependable Computing for Critical Applications*, pages 15–22, 1989. Rapport LAAS No89043
Dependable Computing for Critical Applications, Vol.4, Eds. A.Avizienis, JC.Laprie, Springer Verlag, 1991, pp.23-50.

[111] P. Thévenod-Fosse and H. Waeselynck. An investigation of software statistical testing. *The Journal of Software Testing, Verification and Reliability*, 1(2) :5–26, july-september 1991.

[112] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing : deterministic versus random input generation. *21st IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS'21)*, pages 410–417, 1991.

[113] N. M. Thiéry. Mupad-combinat – algebraic combinatorics package for mupad. http ://mupad-combinat.sourceforge.net/.

[114] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[115] J. Tretmans. Testing labelled transitions systems with inputs and outputs. In *8th International Workshop on Protocols Test Systems*, Évry, France, septembre 1995.

[116] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993. ISBN 0-12-701610-4.

[117] H. Waeselynck. *Vérification De Logiciels Critiques Par Le Test Statistique*. PhD thesis, Institut National Polytechnique, Toulouse, janvier 1993. Rapport LAAS No93006.

[118] L.J. White. *Basic Mathematical Definitions and Results in Testing*, pages 13–24. edited by B. Chandrasekaran and S. Radicchi, North-Holland Publishing Company, 1981. ISBN 0-44-86292-7.

[119] H.S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, (24) :281–291, 1977.

[120] D.B. Wilson. Web site for perfectly random sampling with markov chains. http ://dimacs.rutgers.edu/ dbwilson/exact.

[121] J.B. Wordsworth. *Software Development with Z.* International Computer Science Series. Addison Wesley, 1992.

[122] D.F. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 48–54, 1989.

[123] N. Zuanon. Modular feature integration and validation in a synchronous context. In *Language Constructs for Describing Features, Proceedings of the FIREworks workshop*, pages 213–231, Glasgow, UK, 2000. Springer.

Index

Table des figures

Liste des tableaux

Résumé

Cette thèse propose une nouvelle approche pour le test statistique de logiciel à partir d'une description graphique des comportements du système à tester (graphe de contrôle, statecharts). Son originalité repose sur la combinaison de résultats et d'outils de combinatoire (génération aléatoire de structures combinatoires) et d'un solveur de contraintes, pour obtenir une méthode de test complètement automatisée. Contrairement aux approches classiques qui tirent des entrées, la génération aléatoire uniforme est utilisée pour tirer des chemins parmi un ensemble de chemins d'exécution ou de traces du système à tester. Puis, une étape de résolution de contraintes est utilisée pour déterminer les entrées qui permettront d'exécuter ces chemins. De plus, nous montrons comment les techniques de programmation linéaire peuvent améliorer la qualité d'un ensemble de tests.

Une première application a été effectuée pour le test statistique structurel défini par Thévénod-Fosse et Waeselynck (LAAS) et un prototype a été développé. Des expériences (plus de 10000 réalisées sur quatre fonctions issues d'un logiciel industriel) ont été effectuées pour évaluer notre approche et sa stabilité.

Ces expériences montrent que notre approche est comparable à celle du LAAS, est stable et a l'avantage d'être complètement automatisée. Ces premières expériences nous permettent également d'envisager un passage à l'échelle de notre approche. Plus généralement, ces travaux pourraient servir de base pour une nouvelle classe d'outils dans le domaine du test de logiciel, combinant génération aléatoire de structures combinatoires, techniques de programmation linéaire et résolution de contraintes.

Abstract

In this thesis, we describe a new generic method for statistical testing of software procedures, according to any given graphical description of the behavior of the system under test (control flow graph, statecharts, etc.). Its main originality is that it combines results and tools from combinatorics (random generation of combinatorial structures) with symbolic constraint solving, yielding a fully automatic test generation method. Instead of drawing input values as with classical testing methods, uniform random generation routines are used for drawing paths from the set of possible execution paths or traces of the system under test. Then a constraint resolution step is used for finding actual values for activating the generated paths. Moreover, we show how linear programming techniques may help to improve the quality of test set.

A first application has been performed for structural statistical testing, first defined by Thevenod-Fosse and Waeselynck (LAAS) and a tool has been developed. Some experiments (more than 10000 on four programs of an industrial software) has been made in order to evaluate our approach and its stability.

These experiments show that our approach is comparable to the one of the LAAS, is stable and has the additional advantage to be completely automated. Moreover, these first experiments show also that the method scales up well. More generally, this approach could provide a basis for a new class of tools in the domain of software testing, combining random generation of combinatorial structures, linear programming techniques, and constraint solvers.