RAPPORT DE RECHERCHE

L R I

GENERATION DE TESTS A L'AIDE D'OUTILS COMBINATOIRES : PREMIERS RESULTATS EXPERIMENTAUX

GOURAUD S D

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

07/2003

Rapport de Recherche N° 1364

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Génération de tests à l'aide d'outils combinatoires : premiers résultats expérimentaux

Sandrine-Dominique GOURAUD, gouraud@lri.fr
L.R.I., Université Paris-Sud, bât. 490,
91405 Orsay Cedex, France.

Résumé

Dans un précédent article, nous avons proposé l'utilisation d'outils combinatoires et de résolution de contraintes randomisée, pour générer de manière aléatoire des cas de tests. Cette approche se base sur la couverture d'un critère de test structurel d'une manière similaire au test structurel statistique. Afin d'évaluer la puissance de détection de fautes de notre méthode, nous avons développé un système, appelé AuGuSTe, permettant sa mise en pratique et nous avons entrepris une campagne d'expériences.

Nous présentons dans ce rapport de recherche le prototype AuGuSTe, les expériences menées ainsi que les résutats obtenus.

Mots clés : *génération aléatoire, structure combinatoire, résolution de contraintes, test statistique.*

Abstract

In a previous paper, we proposed a new way of automating statistical testing, based on the combination of uniform generation of combinatorial structures, and of randomized constraints solving techniques, and on a structural test criterion, as in some other works about statistical testing.

For assessing its fault detection power, we now have developed a prototype, named AuGuSTe.

In this technical report, we present AuGuSTe, our experiments and their results.

Keywords : *random generation, combinatorial structures, constraints solving, statistical testing.*

1 Introduction

Dans un précédent article [5], nous avons proposé l'utilisation d'outils combinatoires pour générer de manière aléatoire des cas de tests. Cette approche se base sur la couverture d'un critère de test structurel d'une manière similaire au test structurel statistique [10]. Afin d'évaluer la puissance de détection de fautes de notre méthode, nous avons développé un système, appelé AuGuSTe, permettant sa mise en pratique et nous avons entrepris une campagne d'expériences que nous présentons.

Le principe de cette méthode est le suivant : à partir du graphe de contrôle d'un programme, nous sélectionnons aléatoirement des chemins qui couvrent un critère structurel, puis nous résolvons les prédicats associés à ces chemins afin d'obtenir les données d'entrée qui permettent de les exécuter.

L'originalité de cette approche repose sur les techniques aléatoires qui permettent de choisir les chemins et de résoudre les prédicats. En effet, la sélection des chemins se fait à l'aide d'un outil combinatoire CS[2] qui permet d'engendrer uniformément des structures combinatoires (sortes d'arbre ou de graphe).

La résolution de prédicats est quant à elle basée sur un algorithme de résolution randomisée déjà utilisé dans un outil de génération de test pour Lustre, GATEL[6]. La combinaison de ces techniques nous a permis de proposer une méthode de test statistique complètement automatisée.

Afin de pouvoir évaluer son pouvoir de détection ainsi que sa stabilité du point de vue des tirages, nous avons repris les expériences réalisées pour l'évaluation du test structurel statistique[10].

Les résultats montrent que l'efficacité de notre méthode est comparable à celle du test structurel statistique : elle est stable et efficace dans la détection des erreurs. De plus, une extension à des programmes beaucoup plus gros et à d'autres critères structurels que les critères classiques que sont Tous Les Chemins, Toutes Les Enchaînements et Toutes Les Instructions est possible. En effet, cette méthode est basée sur une génération aléatoire de complexité linéaire en nombre de sommets d'un graphe de contrôle, et elle est complètement automatisée.

L'article est organisé de la façon suivante : en section 2 et 3, nous rappelons notre méthode et nous présentons le prototype développé, en section 4, nous rappelons brièvement le test structurel statistique ainsi que les expériences qui avaient été menées au L.A.A.S. pour son évaluation. En section 5, nous présentons le contexte de nos nouvelles expériences ainsi que les résultats que nous avons obtenus. Enfin, nous concluons notre article par une analyse de ces résultats et une présentation de quelques perspectives.

2 Utiliser des outils combinatoires pour générer automatiquement des tests

Notre méthode de génération de tests est basée sur le tirage aléatoire de chemins d'un programme. Cette méthode est complètement automatique. Elle se base sur le graphe de contrôle du programme, ou sur un sous-ensemble de chemins de ce graphe, et sur une résolution randomisée des prédicats associés aux chemins.

Deux grandes étapes peuvent être distinguées :

1. On effectue un tirage parmi des chemins du graphe de contrôle selon une distribution uniforme en utilisant des outils de génération de structures combinatoires [2]
2. On résout les prédicats associés à ces chemins à l'aide d'un solveur de contraintes randomisé afin d'obtenir des tests provoquant leur exécution.

2.1 Graphe de contrôle et structures combinatoires

Il est classique de représenter la structure d'un programme sous la forme d'un graphe que l'on appelle graphe de contrôle. Chaque sommet de ce graphe représente un bloc d'instructions ou une condition, chaque arc représente un enchaînement possible dans l'exécution du programme.

Notre approche repose sur la remarque [5] qu'un tel graphe peut se traduire automatiquement en une spécification de structures combinatoires [4] et qu'il existe le package CS [2], du logiciel MuPAD [7], qui permet de manipuler de telles structures.

Une spécification de structures combinatoires (ou spécification combinatoire) consiste en un ensemble de règles de production construites à partir d'objets de base appelés `atome` et d'opérateurs appliqués à ces structures. Parmi les opérateurs disponibles, la spécification d'un graphe de contrôle n'en nécessite qu'un sous-ensemble :

- `Union(...)` pour un choix : on l'utilise pour exprimer les choix des instructions conditionnelles *IfThen, IfThenElse, While*
- `Prod(...)` pour une séquence : on l'utilise pour exprimer une séquence simple d'instructions
- `Sequence(...,card=`k`)` pour une itération de taille k d'une structure combinatoire : on l'utilise pour exprimer qu'un morceau de code est appliqué exactement k fois (dans le cas $=$) ou au plus k fois (dans le cas $\leq$).

Il existe une relation forte entre les opérateurs `Prod` et `Sequence` qui est :
`Sequence`(A,`card`$= k) \equiv$ `Prod`$(\underbrace{A, A, ..., A}_{k fois})$

et

$\texttt{Sequence(A,card}\leq k) \equiv \texttt{Union}\ (\epsilon, \texttt{Prod(A)}, \texttt{Prod(A,A)}, ..., \texttt{Prod}(\underbrace{A, A, ..., A}_{k fois}))$

où ϵ est un objet de base de taille 0.

Un mot de taille n d'une telle structure est une suite de n atomes. Dans le cas de la représentation d'un graphe de contrôle par une spécification combinatoire, si les arcs sont représentés par des atomes, alors un mot représente une séquence d'arcs (suite d'enchaînements) et donc un chemin du programme.

À l'aide de l'outil CS, il est possible de tirer de manière aléatoire un mot de longueur égale à n. L'extension des tirages de mots de taille fixe n à des mots de taille inférieure ou égale à n se fait facilement à l'aide d'un arc reliant la source de la spécification combinatoire (ou du graphe de contrôle) à elle-même (comme dans l'exemple 1). Les occurrences de cet arc sont bien entendues complètement ignorées lors de la construction du prédicat associé au chemin du graphe de contrôle sélectionné car dans le programme, il ne correspond à aucun enchaînement possible.

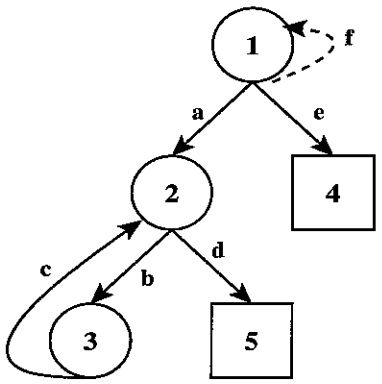

Figure 1:

Exemple 1 (Graphe et spécification combinatoire) *Le graphe de la figure 1 peut-être représenté par la spécification combinatoire ci-dessous où S, NT2, et NT3 sont des non-terminaux :*
S=Union(Prod(*a,NT2*),*e*,Prod(f,S)),
NT2=Union(Prod(*b,NT3*),*d*),
NT3=Prod(*c,NT2*),
a=Atom, *b*=Atom, *c*=Atom, *d*=Atom, *e*=Atom, *f*=Atom,
L'arc f permet de pouvoir étendre la spécification de façon à ne plus considérer des chemins de taille fixe n mais des chemins de taille inférieure ou égale à n. Dans ces cas là, les occurrences de l'arc f seront ignorées.

2.2 Choix de n et des critères de couverture

Afin de pouvoir tirer uniformément dans un ensemble fini de chemins, il nous faut déterminer une taille maximum n des chemins tirables qui peut être très grande. Appelons *chemin élémentaire* un chemin complet du graphe qui va du sommet entrant à un sommet sortant tel que chaque arc n'est emprunté qu'une seule fois. Dans l'exemple 1, il y a 6 chemins élémentaires qui vont du sommet entrant 1 au sommet sortant 3 : `a-b-c-d`, `a-d`, `e`, `f-a-b-c-d`, `f-a-d` et `f-e`. Si n est trop petit, on risque de n'avoir aucun chemin élémentaire voir aucun chemin du tout. Donc n doit être au moins supérieur ou égal à la longueur du plus long chemin élémentaire du graphe de contrôle. Dans le cas ci-dessus, le chemin élémentaire le plus long est `f-a-b-c-d`, donc n est supérieur ou égal à 5.

En présence de boucles, plus n est grand, plus on se rapproche du critère Tous Les Chemins, mais le nombre de chemins considérés croît de manière exponentielle même pour de petits programmes. Beaucoup de tests doivent alors être effectués afin d'obtenir une couverture satisfaisante avec une bonne probabilité.

En effet, si l'on considère la notion de qualité de test définie dans [10] qui représente la probabilité minimale qu'ont les éléments d'un critère d'être atteint lors d'une exécution, alors le nombre de tests N et la qualité de test q_N associée au critère Tous Les Chemins de longueur inférieure ou égale à n sont liés par la relation suivante :

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

où $|C_n|$ représente le nombre de chemins de longueur inférieure ou égale à n. Si on veut une "qualité de couverture" acceptable pour le critère Tous Les Chemins, il faut donc un très grand nombre de tests. Par exemple, en présence de 1000 chemins, il faut au moins 9206 tests pour atteindre une qualité de 0,9999.

2.3 Couvrir Tous LesChemins

Malgré son coût, en l'absence de boucles et pour des programmes moyens, le critère Tous Les Chemins est le plus facile à appliquer. C"est pourquoi nous introduisons notre méthode pour ce critère.

Étant donné une spécification de structures combinatoires représentant le graphe de contrôle du programme à tester, l'algorithme permettant de générer des tests couvrant le critère Tous Les Chemins, se décrit de la façon suivante :

1. Tirer de manière aléatoire et uniforme N chemins
2. Construire les prédicats associés aux N chemins tirés
3. Résoudre en utilisant un solveur de contraintes randomisé les N prédicats

Mais en présence de boucles ou de cas plus complexes, il est nécessaire d'étudier des critères plus faibles comme Tous Les Enchaînements et Toutes Les Instructions.

2.4 Couvrir Tous Les Enchaînements et Toutes Les Instructions

Cette sous-section présente notre approche dans le cas des critères Tous Les Enchaînements et Toutes Les Instructions. Pour abréger la présentation de ces deux critères, qui présentent de nombreuses similitudes, nous appelons *élément* un enchaînement (critère Tous Les Enchaînements) ou un bloc d'instructions (critère Toutes Les Instructions).

Pour comparer les différentes approches de notre méthode pour ces deux critères, nous utilisons la notion de qualité de tests introduite à la section 2.2. Si A est un de ces critères alors la qualité de test q_N associée au critère A et le nombre de tests N sont liés par la relation suivante :

$$(1 - p_{min})^N = 1 - q_N$$

avec $p_{min} = min\{p(e), e \in E_A\}$ où E_A est l'ensemble des éléments du critère A et $p(e)$ est la probabilité qu'une entrée active l'élément e de E_A. Étant donné un nombre de tests fixé N, si l'on veut maximiser q_N alors il faut maximiser les probabilités les plus faibles c'est-à-dire maximiser p_{min}.

Une première approche consiste à suivre le schéma suivant :

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

Nous illustrons cette approche sur un des programmes qui a servi pour nos expériences, le programme FCT2, avec comme critère de couverture Tous Les Chemins.

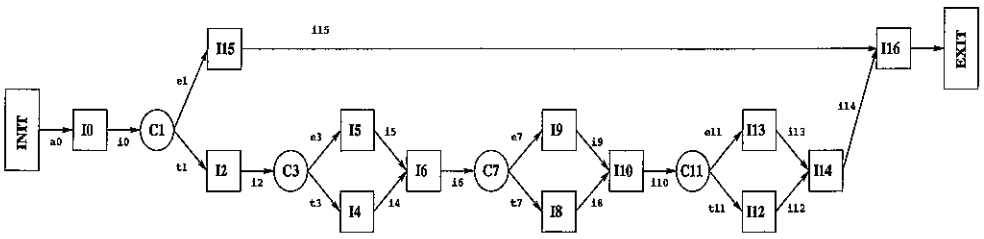

Figure 2: Graphe de contrôle du programme FCT2

Exemple 2 (Le programme FCT2 et le critère Toutes Les Instructions)
D'après la figure 2, l'ensemble E_A des éléments du critère Toutes Les Instructions est le suivant : $E_A = \{I0, I2, I4, I5, I6, I8, I9, I10, I12, I13, I14, I15, I16\}$
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 :*

- $p(I0) = p(I16) = 1$
- $p(I2) = p(I6) = p(I10) = p(I14) =$

  $\frac{1}{13}$ *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 $p(I2) = \frac{106}{117}$
- $p(I4) = p(I5) = p(I8) = p(I9) = p(I12) = p(I13) =$

  $\frac{1}{13}$ *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 $p(I4) = \frac{53}{117}$.
- $p_{min} = p(I15) = \frac{1}{13} + 2 \times \frac{1}{13} \times \frac{1}{9} = \frac{11}{117}$

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.*

L'ensemble E_A bien que naturel ne permet pas toujours de maximiser les probabilités les plus faibles (voir exemple 3).

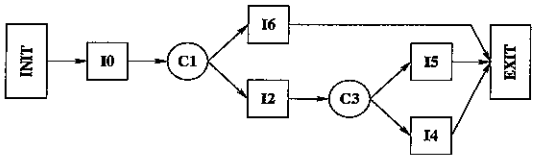

Figure 3:

Exemple 3 (A= Toutes Les Instructions) *Dans le cas de la figure 3,* $E_A = \{I0, I2, I4, I5, I6\}$ *et* $Pmin_{E_A} = p(I6) = \frac{4}{15}$.
Une meilleure solution est de choisir $S = \{I4, I5, I6\}$ *car* $Pmin_S > Pmin_{E_A}$.
En effet, $Pmin_S = p(I4) = \frac{1}{3}$.

Nous proposons de construire un nouvel ensemble S différent de E qui maximise les probabilités les plus faibles. La construction de l'ensemble S et les probabilités des éléments sont intimement liées. En effet, la probabilité d'un élément e est la somme de deux probabilités $p_1(e)$ et $p_2(e)$ qui sont respectivement

la probabilité de tirer l'élément e à l'étape 1 et la probabilité de tirer un chemin lors de l'étape 2 qui passe par e.

$$\begin{cases} p_1(e) = \frac{1}{|S|} \\ p_2(e) = \sum_{e' \in S, e' \neq e} \frac{c^e_{e'}}{c_{e'}} \frac{1}{|S|} \end{cases}$$

où $c_{e'}$ est le nombre de chemins passant par e' et $c^e_{e'}$ est le nombre de chemins passant à la fois par e' et e.

Nous présentons ici la manière dont a été construit l'ensemble S utilisé pour les expériences présentées dans cet article. Cette heuristique est basée sur la classification des éléments en trois classes[5][1] :

- les éléments obligatoires : tous les chemins passent par eux
- les éléments *isolés* : ce sont ceux qui apparaissent dans le moins de chemins
- les éléments obligatoires internes : ce sont ceux qui sont obligatoires dans un sous-graphe donné

L'ensemble S est l'ensemble des éléments isolés avec éventuellement un élément obligatoire. La présence de l'élément obligatoire est liée à la "non couverture" éventuel d'un ou plusieurs chemins. Cette heuristique est meilleure que la solution consistant à prendre tous les éléments du critère mais ne nous permet pas encore de trouver le meilleur S[2].

En résumé, notre approche pour ces critères plus faibles consiste à rajouter une étape à l'algorithme présenté pour le critère Tous Les Chemins. L'algorithme devient alors :

1. Tirer de maniére aléatoire et uniforme N éléments $e_1, ..., e_N$ de S
2. Pour chaque élément tiré e_i, tirer un chemin de manière aléatoire et uniforme parmi tous les chemins du graphe de contrôle qui passent par cet élément
3. Construire les predicats associés aux N chemins
4. Résoudre en utilisant l'algorithme randomisé les N prédicats

L'étape de résolution est effectuée par un solveur de contraintes randomisé. Nous avons repris la bibliothèque développée pour l'outil GATEL [6], et nous l'avons adaptée à nos nouvelles contraintes : en particulier, nous avons rajouté la gestion des tableaux.

3 Le prototype AuGuSTe

Sur ces bases, nous avons développé le prototype AuGuSTe[3]. AuGuSTe travaille sur des programmes simples en langage impératif inspiré de C et Pascal. Les constructions de base sont :

[1] cette classification se base sur la définition de dominance[1]

[2] pour une étude du meilleur choix de S voir[3]

[3] Automated Generation of Structural Statistical Tests

- 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 fixe 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 structuré.

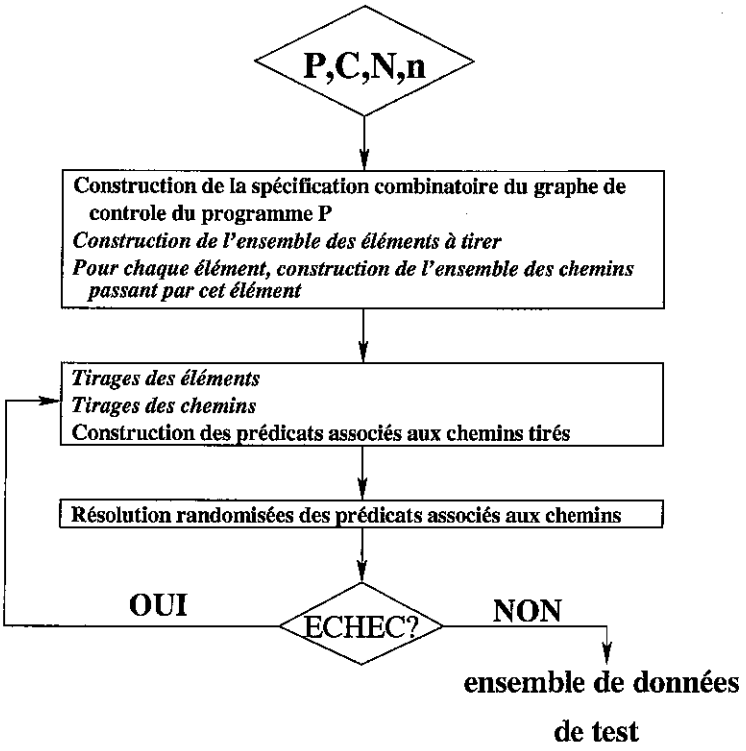

Figure 4: Les différentes étapes du prototype AuGuSTe

Les entrées du prototype sont le programme P à tester, le nombre de tests N désiré, un critère structurel C et une longueur maximale pour les chemins n.

Remarque: n pourrait être automatiquement calculé pour les programmes sans boucle mais ce n'est pas encore implémenté dans AuGuSTe.

Le processus de génération de tests étant complètement automatique, les sorties sont un fichier qui contient les N données de test et un fichier (optionnel) contenant les chemins qui ont été reconnus ou soupçonnés comme étant infaisables.

Le prototype AuGuSTe a trois étapes principales (voir Figure4): une étape de compilation du programme qui construit la spécification combinatoire correspondant au graphe de contrôle du programme; une étape de tirage des éléments et des chemins; une étape de résolution de contraintes qui peut échouer, et dans

ce cas, on retourne à l'étape de tirage. Les phrases en italique dans la figure 4 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.

Concernant les chemins infaisables, il existe plusieurs stratégies possibles. Dans une première version, le prototype AuGuSTe gardait en mémoire les chemins infaisables déjà détectés: lors de l'étape 2, si un chemin tiré était répertorié comme étant un chemin infaisable mémorisé, le prototype AuGuSTe le rejetait aussitôt et en retirait un nouveau.

Malheureusement en présence de trop de chemins infaisables (plus de 4×10^6), le prototype a été confronté à un problème de mémoire. Une deuxième version, celle utilisée pour les expériences, n'utilise aucune stratégie de mémorisation pour la gestion des chemins infaisables.

4 Travaux similaires antérieurs

4.1 La méthode

En 1991, Hélène Waeselynck et Pascale Thévenod-Fosse [8] ont développé une nouvelle forme de test: le test structurel statistique. L'idée de base consistait à pallier les défauts du test structurel en utilisant les avantages du test statistique et les défauts du test aléatoire uniforme en introduisant des aspects structurels.

Leur approche 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.

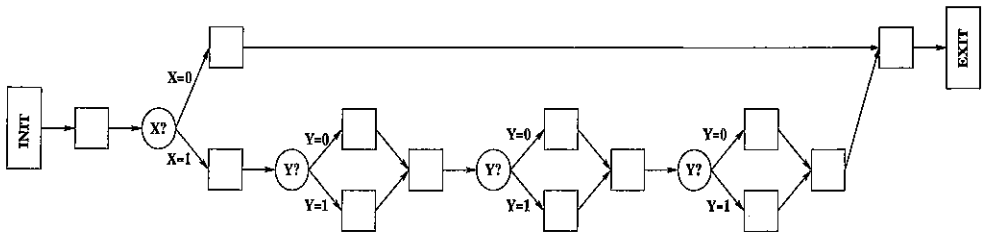

Figure 5: Graphe de contrôle annoté du programme FCT2

Exemple 4 (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 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 Dom(X)={0,1,1,1,1,1,1,1,1} et Dom(Y)={0,1}.

Cette méthode a donné de très bons résultats : 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 : la construction de la distribution d'entrée est essentiellement manuelle, 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.

4.2 Les expériences réalisées

4.2.1 Les programmes testés

Les expériences décrites dans [9] 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 1 récapitule le profil de chacune des 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 point de choix (*While*, *IfThen*, *IfThenElse*).

	♯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

Table 1: Les quatre fonctions testées

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 finis. La fonction FCT4 possède quant à elle une boucle, c'est pourquoi le critère Tous Les Enchaînements a été préféré.

Le nombre de tests nécessaires pour chacune de ces fonctions a été calculé à partir d'une qualité de test égale à 0.9999 ce qui nous donne pour respectivement FCT1, FCT2, FCT3 et FCT4 : 170, 80, 405 et 850 tests. Comme le programme FCT3 est fortement dépendant de l'environnement et le programme FCT4 contient une boucle, cinq séries de jeux de tests ont été effectués pour ces programmes contre une pour les fonctions simples FCT1 et FCT2.

	critère	♯série	♯tests N
FCT1	tous les chemins	1	170
FCT2	tous les chemins	1	80
FCT3	tous les chemins	5	405
FCT4	tous les enchaînements	5	850

Table 2: Nombres de tests réalisés

Remarque : les expériences menées au L.A.A.S. portaient sur une comparaison de différentes méthode de tests, c'est pourquoi une surestimation du nombre de tests nécessaires N apparaît par rapport à l'estimation que nous donnerait la relation entre q_N et N donnée en section 2.

4.2.2 Les expériences du L.A.A.S. à l'aide de ces programmes

Ces tests ont été passés sur 2914 mutants fournis par l'outil SESAME[11]. Pour chaque fonction, le nombre de mutants diffère selon la complexité et la taille de la fonction 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

Table 3: Répartition des mutants pour chaque fonction

Parmi ces mutants, on a classiquement des mutants équivalents. Un mutant équivalent est un mutant tel qu'aucune entrée ne peut le distinguer du programme original. Ces mutants équivalents se divisent en trois catégories[10] en fonction du type de leur équivalence : équivalence fonctionnelle, masquage systématique d'erreur, et équivalence dépendante de l'environnement. Les mutants équivalents de la dernière catégorie peuvent donc changer d'un système à l'autre.

L'efficacité d'un jeu de test pour une fonction FCTn s'évalue à l'aide de son score de mutation c'est-à-dire la proportion de mutants non-équivalents de FCTn qui sont tués, c'est-à-dire détectés, par ce jeu. C'est donc 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.

4.2.3 Les résultats du L.A.A.S.

	♯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 max= 0.9915	6

Table 4: Résultats du test structurel statistique

Le tableau 4 récapitule les résultats obtenus par le test statistique 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.

C'est à partir de ces résultats que nous allons comparer notre nouvelle méthode par rapport au test structurel statistique.

5 Nos expériences

L'objectif des expériences que nous avons menées est double :

- 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.
- comparer les résultats de notre prototype avec ceux obtenus par la méthode du L.A.A.S..

Rappelons que notre prototype AuGuSTe est basé sur le tirage uniforme de chemins et la résolution randomisée de prédicats, alors que la méthode du L.A.A.S. est basée sur la construction d'une distribution sur le domaine des entrées. Dans notre cas, nous ne pouvons pas expliciter 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 donc pas servir de base à une comparaison avec la méthode de la section 4. Par ailleurs, la distribution implicite obtenue est forcément différente de celle du L.A.A.S. puisqu'elle est basée sur des chemins de longueur bornée et ne couvre donc pas tout le domaine d'entrée. C'est le prix à payer pour automatiser en évitant les constructions empiriques que la méthode du L.A.A.S. entraine dans les cas complexes.

5.1 Premiers résultats

Nous avons donc repris les expériences du L.A.A.S. et nous présentons dans cette section les résultats que nous avons obtenus.

Afin de pouvoir évaluer la stabilité de notre méthode, nous avons multiplié par cinq le nombre d'expériences réalisées pour évaluer la méthode de la section 4. Le tableau 5 récapitule nos résultats.

	♯mutants non-équivalents	score de mutation	♯mutants non tués
FCT1	265	1	0
FCT2	548	1	0
FCT3	1416	min=0.9852 max=1 moy=0.9967	entre 0 et 21

Table 5: Résultats expérimentaux pour FCT1, FCT2 et FCT3

Les fonctions FCT1 et FCT2

Ces petites fonctions simples n'ont posé aucun problème. L'ensemble des mutants non-équivalents a été tué.

La fonction FCT3

La présence de variables globales non initialisées rend l'exécution des tests très dépendante de l'environnement et en particulier de l'ordre dans lequel on exécute les tests :

- 15 jeux sur 25 ont obtenu un score de mutation parfait soit 1
- 4 jeux ont obtenu un score de $\frac{1416-7}{1416} = 0.9950$
- 4 jeux ont obtenu un score de $\frac{1416-12}{1416} = 0.9915$
- un jeu a obtenu un score de $\frac{1416-19}{1416} = 0.9866$
- un jeu a obtenu un score de $\frac{1416-21}{1416} = 0.9852$

Ce qui nous donne une moyenne de 0.9967 pour un écart-type de 0.0046. La stabilité de notre approche est très satisfaisante.

5.2 Le cas particulier de FCT4

Le programme FCT4 est le plus intéressant des quatre programmes considérés. En effet, c'est le seul a avoir une boucle et donc une infinité de chemins. Il a soulevé de nombreux problèmes qui nous ont amené à ajuster notre méthode initiale.

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. Dans notre cas particulier, le nombre possible d'itérations de la boucle est limité à exactement 18 ou 19 passages, nous avons donc 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

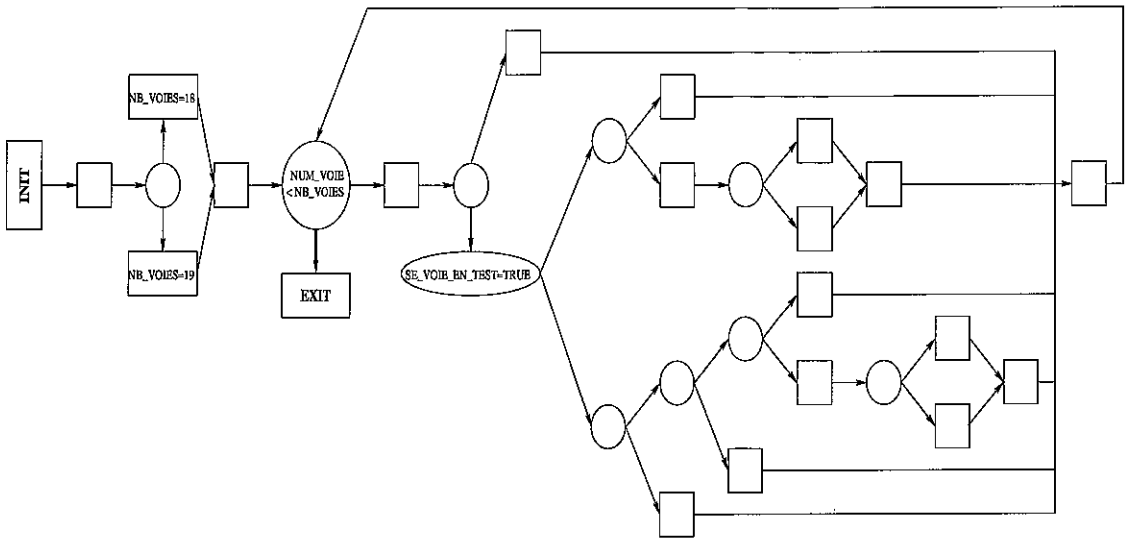

Figure 6: Graphe de contrôle annoté du programme FCT4 et sans code mort

- $L_2 = 12 \times 19$: 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.

Dans un premier temps, la spécification combinatoire ne prenait pas en compte cette limitation du nombre d'itérations ce qui nous donnait 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, nous avons alors intégré la propriété de la boucle dans la spécification combinatoire en utilisant l'opérateur **Sequence** présenté en section 2 et en fixant la cardinalité à 18 et 19. Le nombre de chemins tirables est alors descendu à environ 10^{20} mais avec beaucoup moins de chemins infaisables (tous ceux qui ne passaient pas par exactement 18 ou 19 fois dans la boucle ont été retirés).

Les résultats sont très satisfaisants: parmi les 62 mutants non-équivalents qui n'ont pas été tués, seuls 13 correspondent à un défaut de notre méthode, ce qui nous donne un score de mutation moyen de 0.9762. Les 49 autres mutants correspondent à des mutations que la spécification du programme rend équivalent ou sans intérêt (mutation dans du code mort).

En ce qui concerne les 13 mutants non tués, une analyse des enchaînements couverts montre que certains enchaînements n'ont été couverts par aucun chemin. La non-couverture de ces enchaînements s'explique par la présence de nombreux

chemins infaisables, leur répartition, et leur gestion par le prototype.

L'ensemble des chemins infaisables reste très important car il existe au sein du corps de la boucle un `IfThenElse` dont la condition est une constante (SE_VOIE_EN_TEST=TRUE) qui n'évolue pas avec les itérations.

Dans une première version, le prototype AuGuSTe gérait les chemins dont la résolution avait échoué de la manière suivante: il reprenait l'algorithme en remplaçant N par le nombre de tels chemins. 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.

Une première solution fut de modifier la stratégie de gestion des chemins en échec de la manière suivante:

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.

Une autre solution est d'utiliser les techniques d'optimisation de code. Cela n'est possible que parce que notre programme FCT4 si 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 donc l'expression est invariante[1]. Une telle modification implique une transformation du graphe de contrôle (voir figure 7). Cela n'implique pas nécessairement que la programme qui sera testé soit modifié.

Pour évaluer l'efficacité de notre approche, il nous a fallu reporter cette modification sur les mutants du programme FCT4. Le nombre de mutants est passé de 605 à 724.

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 deux heures, comme pour les autres programmes, contre une semaine lors de nos premiers essais.

Les jeux de tests obtenus ont permis de tuer 6 des 13 mutants non tués des expériences précédentes. Les chemins amenant à l'élimination des 7 derniers

[4]les premiers essais inachevés avaient atteints les 15 jours de recherche sans atteindre la moitié des 850 tests désirés

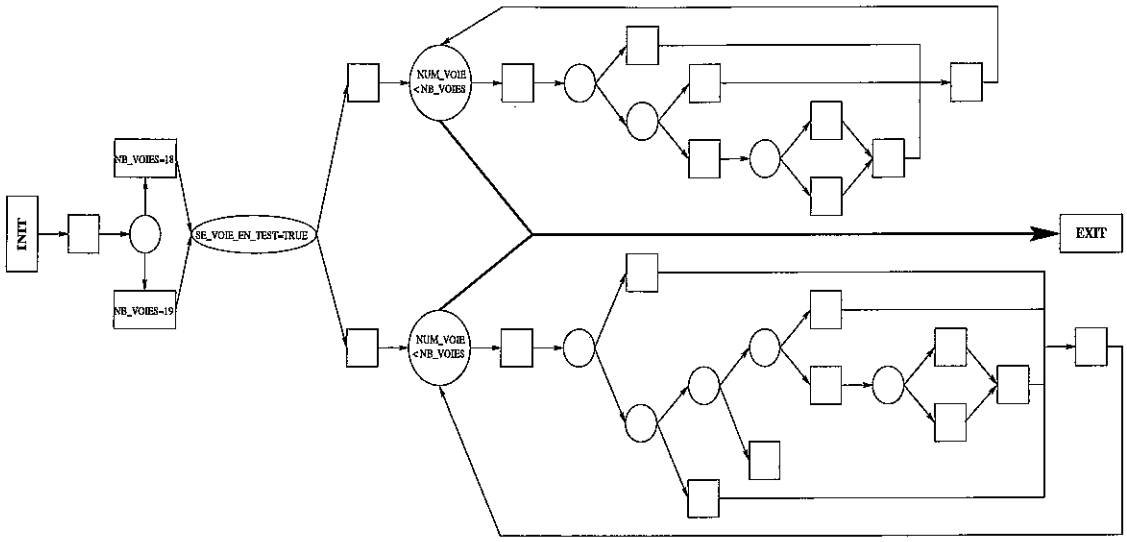

Figure 7: Graphe de contrôle modifié, annoté et sans code mort du programme FCT4

mutants non équivalents restent trop peu probables et seuls trois jeux sur dix sélectionnent un de ces chemins. De nouvelles expériences combinant la modification du graphe et la première solution proposée, c'est-à-dire celle consistant à ne pas retirer dans l'ensemble S, n'ont guère été plus efficaces.

6 Conclusion

Dans cet article, nous avons présenté les expériences réalisées pour évaluer l'efficacité de notre nouvelle approche sur le test statistique. L'originalité de cette méthode repose sur l'utilisation de spécifications de structures combinatoires pour faire du test basé sur une sélection de chemins couvrant un critère structurel. L'utilisation d'outils combinatoires la rend complètement automatisée. À notre connaissance, cela n'avait encore jamais été fait.

Le prototype AuGuSTe est une implémentation de cette méthode qui nous a permis de faire ces expériences. L'objectif des expériences étaient d'évaluer son pouvoir de détection et d'étudier sa stabilité du point de vue statistique. Les expériences ont été réalisées sur quatre programmes venant d'un logiciel industriel et qui avaient déjà servi pour l'évaluation d'une méthode de test similaire, le test structurel statistique.

Les résultats sont très encourageants: la méthode est stable et malgré une heuristique de choix des éléments tirables que nous savons ne pas être la meilleure, les résultats sur la détection des erreurs sont en moyenne très bons. Une formalisation du problème ainsi que l'utilisation de la programmation linéaire pour le

résoudre sont actuellement à l'étude[3]. De plus, les expériences ont révélé la capacité du prototype et de la méthode à traiter des programmes de taille et de complexité significatives : l'ordre de grandeur des prédicats de chemin était d'environ 190 conjonctions et le fait de pouvoir traiter un programme itérant une boucle 18 ou 19 fois nous laisse penser que le traitement de programmes 18 ou 19 fois plus gros est possible.

Étant donné les résultats positifs de nos expériences, un passage à l'échelle de notre méthode ainsi qu'une extension à d'autres techniques de tests comme le test fonctionnel sont donc réalisables. De plus, cette approche statistique, basée sur une génération aléatoire de complexité linéaire en nombre de sommets d'un graphe de contrôle, nous permet d'envisager une automatisation du test "intensif" de programmes critiques.

References

[1] A.Aho, R.Sethi, and J. Ullman. *Compilateurs. Principes, techniques et outils.* IIA, 1989. Interéditions, ISBN 2-7296-0295-X.

[2] S. Corteel, A. Denise, I. Dutour, F. Sarron, and P. Zimmermann. CS web page. http://dept-info.labri.u-bordeaux.fr/~dutour/CS/.

[3] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic tool for statistical testing. Technical report, L.R.I., 2003.

[4] 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.

[5] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *16h I.E.E.E. International Conference on Automated Software Engineering*, pages 5–12, 2001.

[6] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *15h I.E.E.E. International Conference on Automated Software Engineering*, pages 229–237, 2000.

[7] 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/.

[8] 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.

[9] 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.

[10] Hélène Waeselynck. *Vérification De Logiciels Critiques Par Le Test Statistique.* PhD thesis, Institut National Polytechnique, Toulouse, janvier 1993. Rapport L.A.A.S. No93006.

[11] 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 et Maintenabilité*, pages 551–559, 1998.

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1345	FLANDRIN E LI H WEI B	A SUFFICIENT CONDITION FOR PANCYCLABILITY OF GRAPHS	16 PAGES	01/2003
1346	BARTH D BERTHOME P LAFOREST C VIAL S	SOME EULERIAN PARAMETERS ABOUT PERFORMANCES OF A CONVERGENCE ROUTING IN A 2D-MESH NETWORK	30 PAGES	01/2003
1347	FLANDRIN E LI H MARCZYK A WOZNIAK M	A CHVATAL-ERDOS TYPE CONDITION FOR PANCYCLABILITY	12 PAGES	01/2003
1348	AMAR D FLANDRIN E GANCARZEWICZ G WOJDA A P	BIPARTITE GRAPHS WITH EVERY MATCHING IN A CYCLE	26 PAGES	01/2003
1349	FRAIGNIAUD P GAURON P	THE CONTENT-ADDRESSABLE NETWORK D2B	26 PAGES	01/2003
1350	FAIK T SACLE J F	SOME b-CONTINUOUS CLASSES OF GRAPH	14 PAGES	01/2003
1351	FAVARON O HENNING M A	TOTAL DOMINATION IN CLAW-FREE GRAPHS WITH MINIMUM DEGREE TWO	14 PAGES	01/2003
1352	HU Z LI H	WEAK CYCLE PARTITION INVOLVING DEGREE SUM CONDITIONS	14 PAGES	02/2003
1353	JOHNEN C TIXEUIL S	ROUTE PRESERVING STABILIZATION	28 PAGES	03/2003
1354	PETITJEAN E	DESIGNING TIMED TEST CASES FROM REGION GRAPHS	14 PAGES	03/2003
1355	BERTHOME P DIALLO M FERREIRA A	GENERALIZED PARAMETRIC MULTI-TERMINAL FLOW PROBLEM	18 PAGES	03/2003
1356	FAVARON O HENNING M A	PAIRED DOMINATION IN CLAW-FREE CUBIC GRAPHS	16 PAGES	03/2003
1357	JOHNEN C PETIT F TIXEUIL S	AUTO-STABILISATION ET PROTOCOLES RESEAU	26 PAGES	03/2003
1358	FRANOVA M	LA "FOLIE" DE BRUNELLESCHI ET LA CONCEPTION DES SYSTEMES COMPLEXES	26 PAGES	04/2003
1359	HERAULT T LASSAIGNE R MAGNIETTE F PEYRONNET S	APPROXIMATE PROBABILISTIC MODEL CHECKING	18 PAGES	01/2003
1360	HU Z LI H	A NOTE ON ORE CONDITION AND CYCLE STRUCTURE	10 PAGES	04/2003
1361	DELAET S DUCOURTHIAL B TIXEUIL S	SELF-STABILIZATION WITH r-OPERATORS IN UNRELIABLE DIRECTED NETWORKS	24 PAGES	04/2003