Génération de tests “tous-les-chemins” : quelle complexité pour quelles contraintes ?

Nikolai Kosmatov

Génération de tests “tous-les-chemins” : quelle complexité pour quelles contraintes ?*

Nikolai Kosmatov

CEA, LIST, Laboratoire Sûreté des Logiciels, PC 94
91191 Gif-sur-Yvette France
Nikolai.Kosmatov@cea.fr

Résumé

La génération automatique de tests structurels à l'aide de la programmation par contraintes se répand de plus en plus dans l'industrie du logiciel. Parmi les critères de couverture les plus stricts, le critère *tous-les-chemins* exige la génération d'un ensemble de cas de tests tel que tout chemin d'exécution faisable du programme sous test soit exécuté par un des cas de test. Cet article étudie des aspects de calculabilité et de complexité de la génération de tests tous-les-chemins et le rapport entre la complexité et la forme des contraintes issues du programme sous test.

Nous définissons deux classes de programmes importantes pour la pratique. Nous montrons d'abord que pour une classe contenant des programmes simples avec de fortes restrictions, la génération de tests est possible en temps polynomial. Pour une classe de programmes plus large où les entrées peuvent être utilisées comme des indices de tableaux (ou décalages de pointeurs), la génération de tests tous-les-chemins s'avère NP-difficile. Quelques expérimentations montrent le temps de génération pour des exemples de programmes des deux classes.

1 Introduction

La génération automatique de tests à l'aide de la programmation par contraintes passe par la traduction du programme sous test (ou de son modèle formel) et du critère de couverture choisi en un problème de résolution de contraintes. Ensuite, un solveur de contraintes résout les contraintes et fournit *un cas de test*, c'est-à-dire des valeurs pour les variables d'entrée, qui peuvent être accompagnées d'*un oracle* décrivant le comportement attendu du programme sur ces entrées.

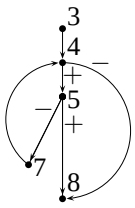
Parmi les méthodes les plus récentes, différentes techniques combinant l'exécution concrète et symbolique sont apparues durant les cinq dernières années. Elles ont permis le développement de plusieurs outils de génération de tests pour les programmes C tels que PathCrawler [2, 14, 15], DART [4], CUTE [12], EXE [3]. Ces techniques se sont montrées particulièrement pertinentes pour le *test orienté chemins*. Par exemple, le critère de couverture *tous-les-chemins* exige la génération d'un ensemble de cas de tests tel que tout chemin d'exécution activable (dit aussi *faisable*) du programme sous test soit exécuté par un cas de test. L'ensemble d'entrées possibles est supposé fini (sinon le nombre de chemins peut être infini, *cf* Section 3). Ce critère étant très fort et souvent inatteignable, des critères plus faibles restreignent la couverture aux chemins de longueur limitée, ou avec un nombre limité d'itérations de boucles, etc. Les chemins sont souvent explorés dans une recherche en profondeur d'abord [4, 12, 14, 15], parfois en largeur d'abord [16] ou avec des heuristiques mixtes [3]. Quand la couverture de chemins s'avère trop exigeante pour le programme sous test, on peut utiliser le critère *toutes-les-instructions* (toute instruction atteignable doit être exécutée par un cas de test) ou *toutes-les-branches* (toute branche atteignable doit être exécutée) [17].

Malgré l'utilisation croissante de la génération automatique de tests en industrie, l'ingénieur validation reste souvent incapable d'évaluer la calculabilité et la complexité de la génération pour un programme donné avec un critère particulier. Les résultats théoriques sous-jacents, souvent difficiles à trouver et à comprendre pour un praticien, traitent le cas le plus général et ne donnent que des réponses négatives. Cependant, les programmes à tester sont en pratique rarement si complexes que le “pire cas” théorique. Une étude détaillée de complexité de la génération de tests pour différents types de programmes en fonction de leurs

*La version originale a été publiée en anglais à *TAIC PART 2009*, Windsor, Royaume-Uni, septembre 2009.

```
1  #define D 4
2  int atU( int x[D], int y[D], int u ) {
3    int i = 1;
4    while( i < D ) {
5      if( u < x[i] )
6        break;
7      i++; }
8    return y[i-1];
9  }
```

(a)



(b)

FIG. 1 – **(a)** Fonction `atU`, et **(b)** son graphe de flot de contrôle

caractéristiques, avec les contraintes et les critères qui en résultent, peut paraître sans intérêt pour un théoricien, mais sera extrêmement utile en pratique.

La motivation de ce travail est d'initier une telle étude de calculabilité et complexité de la génération de tests pour certaines classes de programmes faisant apparaître certains types de contraintes. Dans cet article, nous considérons le cas de la génération de tests tous-les-chemins. Nous présentons d'abord la méthode de génération de tests tous-les-chemins avec une recherche en profondeur d'abord et illustrons son fonctionnement sur un exemple (Section 2).

Contributions. Notre principale contribution consiste à considérer deux classes de programmes. Dans le premier cas (Section 3), grâce à des restrictions sur la taille, le nombre et la forme des contraintes produites, nous montrons que la génération de tests tous-les-chemins en temps polynomial est possible (Théorème 2). La preuve s'appuie sur la méthode polynomiale de résolution des contraintes de différence de Pratt [10, 11]. Strictement parlant, nous prouvons que le temps de résolution de contraintes est polynomial, ce qui est justement la seule étape longue en pratique dans la génération. Nous déduisons la complexité polynomiale pour des critères de couverture plus faibles tels que toutes-les-branches et toutes-les-instructions (Corollaire 3).

Ensuite, la Section 4 considère une classe de programmes plus large pouvant contenir des variables d'entrée dans les indices de tableaux (ou décalages de pointeurs) et contraintes avec $\neq$. Nous donnons une simple preuve par une réduction originale du problème de circuit Hamiltonien que la génération de tests tous-les-chemins pour ces programmes peut être NP-difficile (Théorème 5), donc la génération en temps polynomial est impossible (sauf si P=NP). La Section 5 décrit quelques expériences de génération pour certaines classes de programmes considérées dans les Sections 3 et 4. Enfin, la Section 6 présente la conclusion et les perspectives de travail.

Pour rendre cet article facile à comprendre aussi bien par un spécialiste en théorie de complexité que par un praticien en test de logiciels, nous rappelons des notions des deux domaines, donnons une présentation simplifiée de la génération de tests tous-les-chemins en profondeur d'abord à l'aide de programmation par contraintes, et proposons en Section 4 une ébauche de preuve à l'aide d'un programme C très simple plutôt qu'une preuve formelle sur des machines de Turing. Le lecteur trouvera une introduction à la théorie de calculabilité et complexité dans [5] et plus d'information sur la génération de tests dans [7] et les références dans [7].

2 Génération de tests tous-les-chemins en profondeur d'abord

Cette section décrit la méthode de génération de tests tous-les-chemins à la PathCrawler pour les programmes C, appelée parfois *concolique*. Nous considérons les programmes C avec les types entiers, tableaux, pointeurs (où les variables d'entrée n'apparaissent pas dans les indices de tableaux ou décalages de pointeurs), conditionnels et boucles. Des méthodes similaires sont utilisées dans d'autres outils comme DART [4] et CUTE [12]. Soit f la fonction sous test en C.

L'outil PathCrawler [2, 14, 15] est développé au CEA LIST et comprend deux modules. Le premier, basé sur la bibliothèque CIL [9], traduit les instructions du code source C en contraintes et crée une *version instrumentée* du code qui sera exécutée pour imprimer le chemin d'exécution sur un cas de test. Ensuite, l'utilisateur peut modifier les paramètres de test par défaut et spécifier *une précondition* qui définit les conditions sur les entrées pour lesquelles la fonction f a été conçue et doit être testée. Le second module, *générateur de tests*, implémenté en Prolog, lit les contraintes de f et la précondition et génère des cas de tests pour le critère tous-les-chemins. Les chemins sont explorés en profondeur d'abord. Le générateur utilise une combinaison originale de l'exécution symbolique à l'aide de contraintes et de l'exécution concrète du code instrumenté. L'exécution symbolique traduit le problème de génération de test pour un chemin (partiel) en un problème de résolution de contraintes et appelle un sol-

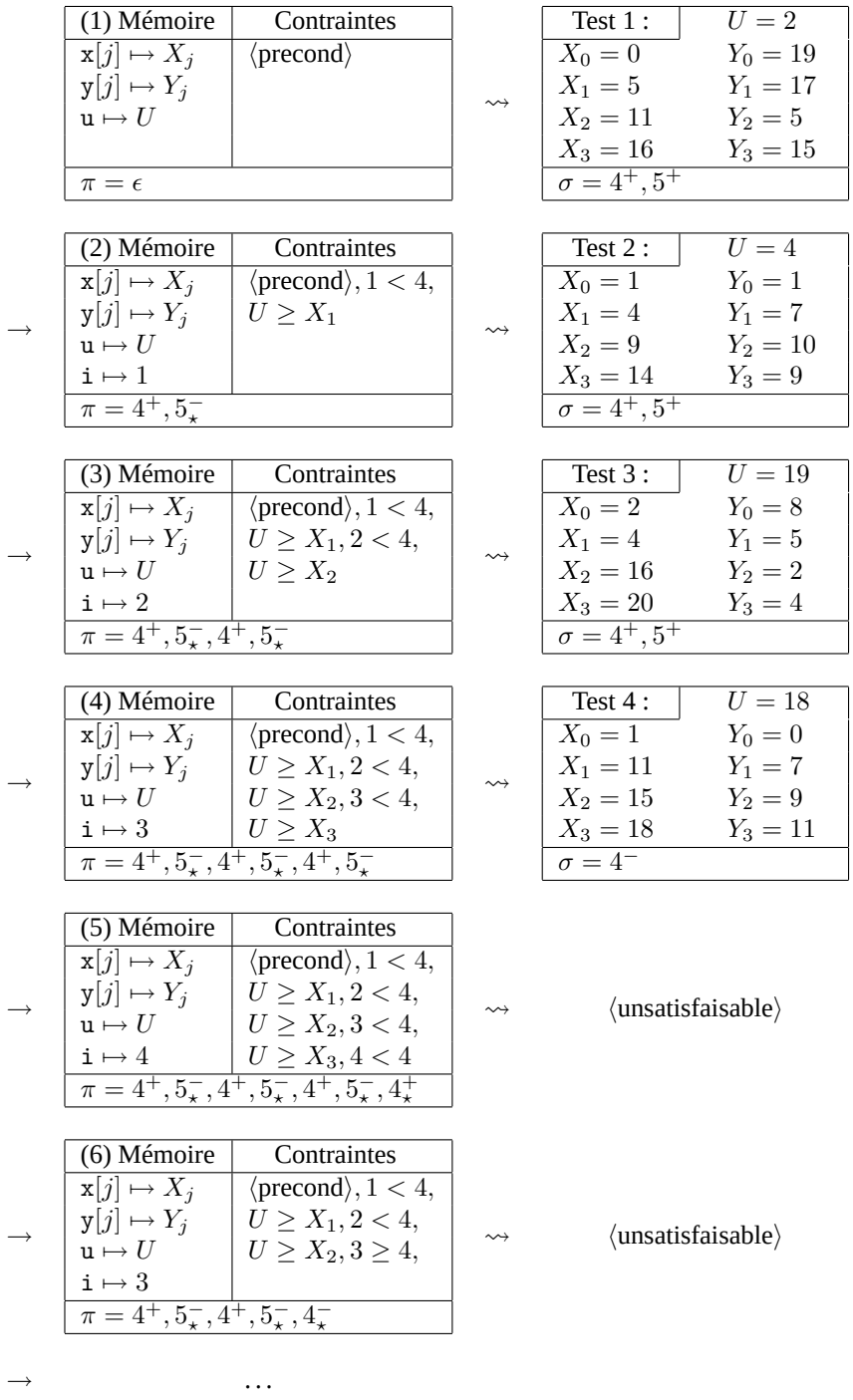


FIG. 2 – Génération de tests tous-les-chemins pour la fonction `atU` de la Figure 1

veur de contraintes pour générer un cas de test activant ce chemin. L'exécution concrète du code instrumenté compilé sur un cas de test fournit rapidement le chemin réellement exécuté. PathCrawler utilise COLIBRI, un solveur performant développé au CEA LIST, qu'il partage avec les outils GATeL [8] et OSMOSE [1].

Nous supposons que le programme sous test a au plus une instruction par ligne et une condition par décision. (Le premier module de PathCrawler réécrit les conditions multiples en conditions simples en introduisant des conditionnels supplémentaires.) Un chemin de programme ρ sera noté par une suite de numéros de lignes, e.g.

$$\rho = 3, 4^+, 5^-, 7, 4^+, 5^+, 6, 8$$

est un chemin dans le programme de la Figure 1. Pour les décisions (dans les conditionnels ou boucles), le numéro de ligne est suivi d'un "$+$" si la condition est vraie, et d'un "$-$" sinon. Puisque un chemin d'exécution est déterminé par ses décisions, nous pouvons abréger un chemin par une suite de décisions, e.g. $\rho = 4^+, 5^-, 4^+, 5^+$. Nous utiliserons cette notation abrégée. Le chemin vide est noté ϵ.

La marque "$\star$" après une décision indiquera que le parcours en profondeur d'abord a déjà complètement exploré sa négation, i.e. l'autre branche dans l'arbre des chemins d'exécution. Par exemple, la marque "$\star$" dans le chemin $\rho = 4^+, 5^-_\star, 4^+, 5^+$ signifie que nous avons déjà exploré tous les chemins de la forme $4^+, 5^+, \ldots$ et essayé de générer un test pour chacun d'entre eux.

Lors d'une session de génération, le générateur maintient les données suivantes :

- un tableau représentant la mémoire du programme à tout moment de l'exécution symbolique. Il peut être vu comme une fonction $Symb \mapsto Val$ qui associe une valeur Val (une constante ou une variable logique Prolog) à un nom symbolique $Symb$ (un nom de variable ou un élément de tableau).
- un chemin partiel π dans f. Si un cas de test est généré pour le chemin partiel π, alors σ notera la partie restante du chemin complet activé par ce cas de test.
- un store de contraintes contenant les contraintes collectées par l'exécution symbolique du chemin partiel courant π.

Nous pouvons maintenant décrire la méthode de génération. Elle comprend les étapes suivantes :

(Init) Pour chaque entrée, on crée une variable logique et l'associe à cette entrée. Pour les variables initialisées, on enregistre leurs valeurs initiales. On pose les contraintes de la précondition. Soit le chemin partiel courant π vide. Passe à l'étape 1.

(Etape 1) Soit σ vide. Le générateur exécute symboliquement le chemin partiel π, i.e. rajoute des contraintes et met à jour la mémoire selon les instructions dans π. Si une contrainte échoue, passe à l'étape 4. Sinon, passe à l'étape 2.

(Etape 2) Le solveur de contraintes est appelé pour générer un cas de test, c'est-à-dire des valeurs pour les variables d'entrée vérifiant les contraintes posées. S'il échoue, passe à l'étape 4. Sinon, passe à l'étape 3.

(Etape 3) La version instrumentée du programme est exécutée sur le cas de test généré à l'étape 2 pour retrouver le chemin complet. Il commence par π (par définition du problème de résolution de contraintes qui a permis de générer ce cas de test). Soit σ la partie restante du chemin complet. Passe à l'étape 4.

(Etape 4) Soit ρ la concaténation de π et σ. Si ρ ne contient aucune décision non marquée par une "$\star$", l'algorithme s'arrête. Sinon, si $x^\pm$ est la dernière décision non marquée dans ρ, soit π le sous-chemin dans ρ avant $x^\pm$, suivi de $x^\mp_\star$ (i.e. la négation de $x^\pm$ marquée comme traitée). Passe à l'étape 1.

Ainsi, l'étape 4 fait une recherche en profondeur d'abord pour choisir le chemin partiel suivant. Elle nie la dernière condition non marquée dans ρ pour trouver les différences les plus profondes d'abord, et marque une décision par une "$\star$" quand sa négation (i.e. l'autre branche partant de ce nœud dans l'arbre des chemins d'exécution) est complètement explorée. Par exemple, si

$$\rho = a^-_\star, b^+, c^+, d^-_\star, e^+_\star,$$

la dernière "$\star$" signifie que la recherche en profondeur d'abord a déjà exploré tous les chemins de la forme

$$a^-, b^+, c^+, d^-, e^-, \ldots$$

La précédente "$\star$" (dans $d^-_\star$) signifie que la recherche en profondeur d'abord a déjà traité tous les chemins

$$a^-, b^+, c^+, d^+, \ldots$$

La dernière décision non marquée dans ρ étant c^+, l'étape 4 prendra le sous-chemin avant cette décision $a^-_\star, b^+$ et rajoutera $c^-_\star$ pour obtenir le nouveau chemin partiel $\pi = a^-_\star, b^+, c^-_\star$. Ainsi, ce marquage de conditions avec une "$\star$" préserve les informations sur le parcours de des chemins partiels plus courts (ici, les chemins de la forme $a^+, \ldots$ sont complètement explorés), et rajoute ces informations pour la négation de la dernière condition (ici, les chemins $a^-, b^+, c^+, \ldots$ sont complètement explorés).

Nous déroulons cette méthode sur l'exemple de la fonction `atU` de la Figure 1. Cette fonction est la forme la plus simple d'interpolation. Elle prend trois paramètres, deux tableaux `x`, `y` (chacun avec `D` entiers) et un entier `u`. Soit $\psi_{\texttt{atU}}$ la précondition de `atU` définie comme suit :

$$\begin{array}{c} \texttt{D} \geq 1, \quad \texttt{x} \text{ contient } \texttt{D} \text{ éléments}, \\ \texttt{y} \text{ contient } \texttt{D} \text{ éléments}, \\ 0 \leq \texttt{x}[0] < \texttt{x}[1] < \cdots < \texttt{x}[\texttt{D}-1] \leq \mathrm{Max}, \\ \texttt{x}[0] \leq \texttt{u} \leq \texttt{x}[\texttt{D}-1], \\ 0 \leq \texttt{y}[0], \texttt{y}[1], \ldots, \texttt{y}[\texttt{D}-1] \leq \mathrm{Max}. \end{array} \qquad (\psi_{\texttt{atU}})$$

On suppose que les $\texttt{y}[j]$ sont les valeurs d'une certaine fonction h en les points $\texttt{x}[j]$, i.e. $h(\texttt{x}[j]) = \texttt{y}[j]$, $0 \leq j \leq \texttt{D}-1$. La fonction `atU` retourne la valeur de h en le point le plus proche à gauche de `u` pour lequel la valeur de h est connue. Autrement dit, elle trouve le plus grand k tel que $\texttt{x}[k] \leq \texttt{u}$ et retourne $\texttt{y}[k]$. Max est une constante positive (e.g. la valeur maximale entière du système). Pour simplifier notre exemple, on suppose $\texttt{D} = 4$ et $\mathrm{Max} = 20$.

La session de génération de tests pour la fonction `atU` est présentée en Figure 2, où "$\rightsquigarrow$" note l'application des étapes 2 et 3, et "$\rightarrow$" l'application des étapes 4 et 1.

D'abord, (Init) crée des variables logiques X_j, Y_j ($0 \leq j \leq 3$) et U pour représenter les entrées, *cf* (1) de la Figure 2. Les deux premières lignes de ($\psi_{\texttt{atU}}$) étant satisfaites, (Init) ajoute dans le store de contraintes les $3\texttt{D}+3$ inégalités correspondant aux trois dernières lignes de ($\psi_{\texttt{atU}}$), qui sont notées par $\langle$precond$\rangle$ dans la Figure 2 :

$$\begin{gathered} 0 \leq X_0,\ X_0 < X_1,\ X_1 < X_2,\ X_2 < X_3,\ X_3 \leq 20, \\ X_0 \leq U,\ U \leq X_3, \\ 0 \leq Y_0,\ \ldots\ ,\ 0 \leq Y_3, \\ Y_0 \leq 20,\ \ldots\ ,\ Y_3 \leq 20. \end{gathered}$$

Le premier chemin partiel π étant toujours vide, l'étape 1 n'a rien à faire ici. Ensuite, l'étape 2 génère le premier cas de test, Test 1. L'étape 3 exécute le Test 1 sur la version instrumentée du programme et obtient $\sigma = 4^+, 5^+$ (ce qui abrège $3, 4^+, 5^+$).

Nous passons maintenant de (1) et Test 1 à (2) dans la Figure 2. L'étape 4 calcule $\rho = 4^+, 5^+$, où 5^+ est la dernière décision non marquée. Par conséquent, $\pi = 4^+, 5^-_\star$. Ensuite, l'étape 1 exécute symboliquement le chemin partiel π en contraintes, nœud par nœud, avec des entrées inconnues. L'exécution de l'affection 3 ajoute $\texttt{i} \mapsto 1$ dans la mémoire. L'exécution de la décision 4^+ ajoute la contrainte $1 < 4$ trivialement vraie, et l'exécution de la décision 5^- ajoute la contrainte $U \geq X_1$ après avoir remplacé les variables i, u et $\texttt{x}[1]$ par leurs valeurs actuelles dans la mémoire. Ensuite, l'étape 2 génère le Test 2, et l'étape 3 l'exécute et obtient $\sigma = 4^+, 5^+$.

Nous passons maintenant de (2) et Test 2 à (3) dans la Figure 2. L'étape 4 calcule $\rho = 4^+, 5^-_\star, 4^+, 5^+$ et déduit $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star$. L'étape 1 exécute symboliquement π, l'étape 2 génère le Test 3, etc. Arrêtons-nous sur le passage de (4) et Test 4 à (5) dans la Figure 2. L'étape 4 calcule

$$\rho = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-$$

et déduit $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+_\star$. La dernière contrainte $4 < 4$ ajoutée par exécution symbolique à l'étape 1 est fausse, donc le générateur passe directement à l'étape 4, qui pose $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-_\star$. Selon (6) de la Figure 2, la dernière contrainte $3 \geq 4$ ajoutée par l'étape 1 échoue de nouveau, et le générateur passe à l'étape 4. Les Etape après (6) ne sont pas détaillées en Figure 2. De même, le générateur essaie les chemins partiels $\pi = 4^+, 5^-_\star, 4^-_\star$ et $\pi = 4^-_\star$, qui sont infaisables, et s'arrête. Un cas de test est généré pour chacun des 4 chemins faisables.

En général, si pendant l'exécution symbolique aux étapes 1 ou 2, les contraintes sont insatisfaisables et aucun cas de test ne peut être généré, alors π est infaisable et l'algorithme continue l'exploration des autres chemins normalement. Si cela se produit à (Init) or à la première itération de l'étape 2, i.e. la précondition est insatisfaisable, alors l'algorithme s'arrête à l'étape 4 car ρ est vide.

3 Génération de tests tous-les-chemins en temps polynomial

Cette section présente des conditions suffisantes sous lesquelles une classe de problèmes de génération de tests tous-les-chemins devient polynomiale. Il est intuitivement clair que la génération de tests tous-les-chemins pour un programme peut prendre beaucoup de temps pour une (ou plusieurs) des raisons suivantes :

- (†) le programme a un très grand nombre de chemins, et se traduit donc par un très grand nombre de problèmes de résolution de contraintes,
- (††) les instructions du programme se traduisent par des contraintes complexes qui ne peuvent être résolues rapidement,
- (†††) le programme a de très longs chemins faisant apparaître des problèmes avec trop de contraintes.

Nous montons que des restrictions appropriées sur ces trois facteurs assurent la génération de tests tous-les-chemins en temps polynomial.

Puisque l'étape la plus coûteuse de la génération de tests tous-les-chemins en pratique est la résolution de contraintes à l'étape 2, nous allons nous focaliser sur le temps de cette résolution. Notre expérience avec l'outil PathCrawler montre que les autres étapes (instrumentation, traduction en contraintes, exécution symbolique etc.) sont très efficaces. Par exemple, le premier module de PathCrawler, qui instrumente et traduit en contraintes le programme sous test, met moins d'une minute pour des programmes de plusieurs centaines et milliers de lignes de code. Comme la performance de ces étapes dépend de nombreux détails d'implémentation et semble très satisfaisante en pratique, nous la laissons de côté ici. Notons que le temps de génération de tests dans les expérimentations de la Section 5 comprend toutes les étapes, depuis le code source jusqu'aux cas de test générés.

Nous définissons *un problème de génération de tests tous-les-chemins* comme

$$\Phi = (P, f, \psi)$$

où P est un programme, f est une fonction à tester dans P et ψ est la précondition de f. Une solution de Φ est un ensemble de cas de test satisfaisant le critère tous-les-chemins. La précondition peut contenir des informations nécessaires pour une initialisation correcte de la génération de tests (e.g. tailles de tableaux d'entrée, domaines de variables, etc.) et toute autre conditions sur les variables d'entrée réduisant les entrées admissibles de f. Nous notons par $L^\Phi_P > 0$ la taille du programme P.

Le nombre d'entrées possibles doit être fini (et non pas seulement borné par la mémoire disponible de l'ordinateur qui est supposée suffisamment grande). En effet, si le nombre d'entrées est illimité, le nombre de chemins peut

```
1  char LastChar(char str[]) {
2    while( *(str + 1) != 0 )
3      str = str + 1;
4    return * str;
5  }
```

FIG. 3 – Pour une chaîne de caractère non vide terminée par un 0, `LastChar` retourne son dernier caractère non nul

être illimité et la génération de tests ne terminera pas. Cela se produit pour la fonction `LastChar` de la Figure 3.

Nous supposons donc que ψ borne la taille maximale des entrées admissibles pour Φ (mesurée en octets, ou à une constante près, en entiers) par un $L_I^\Phi > 0$.

Un système de contraintes de différence [11] est défini comme un système de contraintes de la forme

$$x - y \leq c, \quad \text{ou} \quad x \leq c, \quad \text{ou} \quad x \geq c,$$

où x, y sont des variables entières et c est un entier. Une égalité $x - y = c$ peut être représentée comme $x - y \leq c$ et $y - x \leq -c$.

Théorème 1 ([10, 11]) *Un système de contraintes de différence peut être résolu en temps polynomial $g(m, n)$, où $g(X, Y)$ est un polynôme, n le nombre de variables et m le nombre de contraintes.*

Le lecteur trouvera plusieurs estimations g dans [11]. Nous sommes prêts à énoncer le résultat principal de cette section. Notez que les conditions (i), (ii), (iii) correspondent précisement aux facteurs (†), (††), (†††).

Théorème 2 *Soit $\mathcal{C}$ une classe de problèmes de génération de tests tous-les-chemins, et soient $g_1(X, Y)$, $g_2(X, Y)$ deux polynômes. On suppose que tout problème $\Phi = (P, f, \psi)$ de $\mathcal{C}$ satisfait les propriétés suivantes :*

(i) *le nombre de chemins dans Φ pour lesquels la méthode essaiera de générer un cas de test est borné par $g_1(L_P^\Phi, L_I^\Phi)$,*

(ii) *l'exécution symbolique de chaque chemin (y-compris les contraintes de la précondition) ajoute uniquement des contraintes de différence,*

(iii) *l'exécution symbolique de chaque chemin (y-compris les contraintes de la précondition) ajoute au plus $g_2(L_P^\Phi, L_I^\Phi)$ contraintes.*

Alors il existe un polynôme $g_3(X, Y)$ tel que le temps de résolution de contraintes total lors de la génération de tests tous-les-chemins pour Φ est borné par $g_3(L_P^\Phi, L_I^\Phi)$.

Ebauche de preuve. Sans perte de généralité, on suppose $g(X, Y)$ monotone en chaque argument pour $X > 0, Y > 0$. Soit $\Phi = (P, f, \psi)$ un problème de génération de tests tous-les-chemins dans $\mathcal{C}$. Selon (i), la méthode de génération de tests tous-les-chemins pour Φ résout au

```
1  #define D 4
2  int bsearch(int a[D], int key) {
3    int low = 0; int high = D-1;
4    while (low <= high) {
5      int mid = low + (high-low)/2;
6      int midVal = a[mid];
7      if (midVal < key)
8        low = mid+1;
9      else if (midVal > key)
10       high = mid-1;
11     else
12       return mid;
13   }
14   return -1;
15 }
```

FIG. 4 – Etant donné un élément `key` et un tableau trié `a` de taille `D`, `bsearch` fait une recherche binaire de `key` dans `a`

plus $g_1(L_P^\Phi, L_I^\Phi)$ problèmes de résolution de contraintes. D'après (ii), le problème de résolution de contraintes créé pour chaque chemin (donc, chaque chemin partiel) de Φ est un système de contraintes de différence. D'après (iii), ce sytème contient $m \leq g_2(L_P^\Phi, L_I^\Phi)$ contraintes. Le nombre de variables n est borné par L_I^Φ. Il suit que le temps de résolution total est borné par

$$g_1(L_P^\Phi, L_I^\Phi) g(m, n) \leq g_3(L_P^\Phi, L_I^\Phi),$$

où $g_3(X, Y) := g_1(X, Y) g(g_2(X, Y), Y)$. □

Nous permettons intentionnellement de borner le nombre de chemins par la taille du programme, ou la taille des entrées, ou les deux, car différentes estimations peuvent être utiles dans différents exemples. Notons que le nombre de chemins mentionnés dans (i) inclut les chemins partiels infaisables comme ceux vus dans la Section 2, mais n'inclut pas plusieurs chemins commençant par le même chemin partiel infaisable. En effet, la méthode en profondeur d'abord rajoute au plus une nouvelle contrainte aux contraintes d'un chemin partiel faisable, donc elle n'essaie jamais de générer un cas de test pour un chemin plus long qui contient un chemin partiel infaisable strictement plus court.

Appliquons le Théorème 2 à un exemple. Soit $\Phi_{\mathtt{D}} = (P_{\mathtt{D}}, \mathtt{atU}, \psi_{\mathtt{atU}})$ la famille de problèmes de génération de tests tous-les-chemins, où $\mathtt{D} > 0$ est un paramètre et $P_{\mathtt{D}}$ est le programme de la Figure 1 avec la fonction `atU` et la précondition $\psi_{\mathtt{atU}}$ définie dans la Section 2. Le nombre de variables d'entrée dans $\Phi_{\mathtt{D}}$ est $2\mathtt{D} + 1$, donc $L_I^{\Phi_{\mathtt{D}}} = 2\mathtt{D} + 1$. Le nombre de chemins partiels dans $\Phi_{\mathtt{D}}$ pour lesquels la méthode essaiera de générer un cas de test est égal à $2\mathtt{D} \leq L_I^{\Phi_{\mathtt{D}}}$, d'où (i). L'exécution symbolique des chemins de $\Phi_{\mathtt{D}}$ n'ajoute que des contraintes de différence, d'où (ii), avec $3\mathtt{D} + 3$ contraintes pour la précondition et

$2\mathtt{D}-1$ contraintes pour le chemin le plus long, d'où (iii) car $(3\mathtt{D}+3)+(2\mathtt{D}-1) \leq 3L_I^{\Phi_\mathtt{D}}$. Les conditions du Théorème 2 sont vérifiées, donc la génération de tests tous-les-chemins en temps polynomial est possible pour cette exemple.

De la même façon, ce théorème peut être appliqué à d'autres fonctions d'interpolation utilisées en pratique, ou à des fonctions de recherche dans un tableau comme la fonction `bsearch` de la Figure 4. Etant donné un tableau trié `a` de taille `D` et un élément `key`, la fonction `bsearch` effectue une recherche binaire (dichotomique) classique d'une occurence de `key` dans `a` et retourne son indice, ou -1 si `key` n'apparaît pas dans `a`. A première vue, l'affectation de la ligne 5 de la Figure 4 n'ajoute pas une contrainte de différence. En réalité, pendant l'exécution symbolique d'un chemin partiel, la partie droite des affectations lines 3, 5, 8, 10 contient uniquement des constantes sans variables d'entrée, qui permettent un calcul direct de la nouvelle valeur de la variable et n'ajoutent pas de contrainte sur des variables d'entrée.

Puisque le critère tous-les-chemins est plus fort que d'autres critères de couverture comme toutes-les-branches ou toutes-les-instructions [17], le résultat suivant découle directement du Théorème 2.

Corollaire 3 *Soit $\mathcal{C}$ une classe de problèmes de génération de tests tous-les-chemins satisfaisant les conditions du Théorème 2. Alors le temps total de résolution de contraintes pour la génération de tests avec le critère toutes-les-branches (ou toutes-les-instructions) est polynomial.*

4 Génération de tests tous-les-chemins avec alias internes est NP-difficile

Dans cette section nous allons considérer une classe plus large de problèmes de génération de tests tous-les-chemins où les contraintes avec $\neq$ sont autorisées et les indices de tableaux (ou décalages de pointeurs) peuvent dépendre des variables d'entrée. La présence d'indices inconnus pendant l'exécution symbolique pour des entrées inconnues nous conduit au problème des *alias internes*, définis dans [6]. En effet, si `j` est une variable d'entrée, ou a reçu une valeur dépendant de variables d'entrée, l'expression `a`$[$`j`$]$ est un alias non-trivial à un des aléments de `a`. L'utilisation des entrées `p`$[$`j`$]$ comme indices du tableau `G` lines 26, 28 de la Figure 5a est un autre exemple d'alias internes. La génération de tests tous-les-chemins pour les programmes avec des alias internes a été considérée dans [6] où une extension de la méthode de la Section 2 pour ce type de programmes a été proposée.

Rappelons le problème du circuit Hamiltonien. *Un circuit Hamiltonien* dans un graphe orienté G est un circuit qui passe par chaque nœud une seule fois et revient vers le nœud de départ. Par exemple, la Figure 5b représente un graphe orienté avec 5 nœuds $\{0, 1, 2, 3, 4\}$ ayant un circuit Hamiltonien. On peut identifier un graphe orienté avec sa *matrice d'adjacence*. Son élément $G(i, j)$ est 1 si G a un arc allant de i à j, et 0 sinon. Nous préférons ici la notation mathématique $G(i, j)$, $p(i)$ à la notation C (en police TrueType) `G`$[$`i`$][$`j`$]$, `p`$[$`i`$]$.

Dans les lignes 5–11 de la Figure 5a, G est le graphe de la Figure 5b (avec $N = 5$ nœuds) défini par sa matrice d'adjacence. La première boucle dans la fonction HC vérifie que les éléments de p sont dans $\{0, 1, \ldots, N-1\}$ et sont tous différents (lines 15–23). Cela signifie que p est *une permutation de* $\{0, ..., N-1\}$. HC retourne 1 si p est une permutation des nœuds de G et

$$p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$$

est un circuit Hamiltonien dans G, et 0 sinon (lignes 25–32). La précondition ψ_{HC} est définie comme suit :

$$\begin{array}{c} p \text{ contient } N \text{ éléments,} \\ 0 \leq p(j) \leq MAXINT. \end{array} \qquad (\psi_{\mathrm{HC}})$$

Nous allons admettre la Conjecture 4, conséquence de la fameuse conjecture $P \neq NP$ que l'on croit vraie, et formuler le résultat principal de cette section.

Conjecture 4 ([5, Section 10.4.4]) *Il n'existe pas d'algorithme décidant en temps polynomial si un graphe orienté donné contient un circuit Hamiltonien.*

Théorème 5 *Il n'existe pas d'algorithme polynomial pour résoudre les problèmes de génération de tests tous-les-chemins pour les programmes avec des alias internes.*

Ebauche de preuve. Supposons le contraire. Soit A un algorithme polynomial qui, étant donné un problème de génération de tests $\Phi = (P, f, \psi)$, génère une liste

$$(t_1, \rho_1), \ldots, (t_k, \rho_k)$$

où t_i est un cas de test, ρ_i est le chemin d'exécution activé par l'exécution de f sur t_i, et $\rho_1, \ldots, \rho_k$ sont tous les chemins faisables de f. "Un algorithme polynomial" signifie qu'il existe $K, m > 0$ tels que le nombre de pas de A est toujours borné par le polynôme $K(L_P^\Phi + L_I^\Phi)^m$, où L_P^Φ est la taille de P et L_I^Φ est la taille maximale des entrées.

Alors nous pouvons construire un nouvel algorithme B qui, étant donné la matrice d'adjacence G d'un graphe orienté et son nombre de nœuds N,

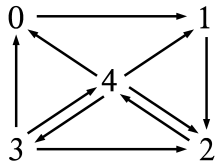
$B1$) construit un programme P_G similaire à celui de la Figure 5a avec les N et G donnés,

$B2$) exécute A sur le problème $\Phi_G = (P_G, \mathrm{HC}, \psi_{\mathrm{HC}})$,

$B3$) dit "oui" si A a généré un cas de test pour le chemin retournant 1, et "non" sinon.

```
#define N 5 // number of vertices in graph G
typedef int graph[N][N];
typedef int perm[N];
// graph G is defined by its adjacency matrix :
graph G = {
  0,1,0,0,0,
  0,0,1,0,0,
  0,0,0,0,1,
  1,0,1,0,1,
  1,1,1,1,0
};

int HC(perm p){
  int i, j;
  for( i = 0; i < N; i++ ){
    if( p[i] < 0 )
      return 0;
    if( p[i] > N-1 )
      return 0;
    for( j = i+1; j < N; j++ )
      if( p[i] == p[j] )
        return 0;
  }
// we checked that p is a permutation of {0,...,N-1}
  for( i = 1; i < N; i++ )
    if( G[ p[i-1] ][ p[i] ] != 1 )
      return 0;
  if( G[ p[N-1] ][ p[0] ] != 1 )
    return 0;
// we checked that p defines the Hamiltonian cycle
// p(0) -> p(1) -> ... -> p(N-1) -> p(0) in G
  return 1;
}
```



(a) **(b)**

FIG. 5 – **a)** Etant donné un graphe G avec N nœuds, statiquement défini par sa matrice d'adjacence, la fonction HC vérifie si p est une permutation de nœuds définissant le circuit Hamiltonien $p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$.
b) Le graphe G a le circuit Hamiltonien $0 \to 1 \to 2 \to 4 \to 3 \to 0$.

La taille des entrées (N, G) de B est proportionnelle à N^2 :

$$|G| \leq |(N, G)| \leq 2|G|,\ |G| \sim N^2.$$

Montrons que B est un algorithme polynomial. En effet, pour certains $K_j > 0$, $B1$ copie des chaînes de caractères de taille $\leq K_1 N^2$. $B2$ exécute l'algorithme A sur le problème Φ_G avec $(L_P^\Phi + L_I^\Phi) \leq K_2 N^2$, donc il fait $\leq K(K_2 N^2)^m$ pas. La fonction HC a $\leq K_3 N^2$ chemins faisables, et la taille de chaque chemin est $\leq K_4 N^2$. $B3$ lit la liste de cas de test générés qui peut contenir $\leq K_3 N^2$ couples (t_i, ρ_i), chacun de taille $\leq K_5 N^2$. Donc B fait $\leq K_6 N^{2m} + K_7 N^4 + K_8 N^2$ pas.

Il est clair que le chemin retournant 1 dans la fonction HC de P_G est faisable si et seulement si G a un circuit Hamiltonien. On déduit que B est un algorithme polynomial décidant si un graphe donné G a un circuit Hamiltonien. La contradiction avec la Conjecture 4 termine la preuve. □

L'intérêt du Théorème 5 réside dans sa généralité : il est vrai même pour les programmes les plus simples, comme celui de la Figure 5a, où

- le nombre de chemins est borné par un polynôme en la taille du programme, *cf* (†),
- la longueur des chemins est bornée par un polynôme en la taille du programme aussi, *cf* (†††),
- la fonction sous test f contient seulement des entiers,

	atU		bsearch	
D	tests	temps	tests	temps
4	4	0.50 s	9	0.49 s
10	10	0.52 s	21	0.56 s
50	50	1.38 s	101	2.91 s
100	100	6.06 s	201	12.10 s
500	500	5 m 39 s	1001	6 m 50 s
1000	1000	30 m 46 s	2001	32 m 47 s

FIG. 6 – Résultats de génération de tests tous-les-chemins pour les fonctions `atU` de la Figure 1 et `bsearch` de la Figure 4 pour différentes valeurs de `D`.

	HC		
N	chemins	tests	temps
4	15	38	0.66 s
5	21	140	2.10 s
6	28	747	19.75 s
7	36	5 075	4 m 33 s
8	45	40 364	26 m 58 s
9	55	362 934	4 h 2 m 24 s

FIG. 7 – Résultats de génération de tests tous-les-chemins pour la fonction HC de la Figure 5a sur des graphes complets $\mathrm{G_N}$.

tableaux, conditionnels, affectations et boucles avec un nombre fixe d'itérations,

– f ne contient pas d'appels de fonction, ni d'*alias externes* (qui apparaissent quand f contient des pointeurs en entrée qui peuvent faire référence à certaines cases mémoire de deux façons différentes, *cf* [6]).

Dans cet exemple, la complexité est due à $(\dagger\dagger)$, i.e. la forme des contraintes incluant des alias internes et $\neq$, bien que (i), (iii) du Théorème 2 soient vérifiés.

Remarque. En fait, nous avons montré que le problème de génération de tests tous-les-chemins pour ces programmes est NP-difficile, c'est-à-dire au moins aussi difficile que le problème du circuit Hamiltonien ou tout autre problème NP-complet. Un spécialiste en théorie de complexité remarquera que notre ébauche de preuve peut être transformé en une preuve complète car un ordinateur peut être simulé par une machine de Turing en temps polynomial et inversement [5, Section 8.6]. Une représentation appropriée pour N et p va résoudre le problème de grandes valeurs dépassant la taille du mot sur notre ordinateur.

5 Expérimentations

Cette section présente des expérimentations avec l'outil PathCrawler qui montrent les résultats de génération de tests tous-les-chemins pour des exemples de classes de programmes considérées dans les Sections 3 et 4. Les expériences ont été faites sur un portable Intel Core 2 Duo avec 1Gb RAM.

La Figure 6 montre les résultats expérimentaux pour deux fonctions, `atU` de la Figure 1 et `bsearch` de la Figure 4, pour différentes valeurs du paramètre `D`. Pour chaque `D`, les colonnes "tests" et "temps" montrent le nombre de cas de test générés et le temps de la génération. Ici, PathCrawler génère exactement un cas de test pour chaque chemin faisable, donc le nombre de chemins faisables est égal au nombre de cas de test.

Nous voyons que le temps de génération avec l'outil PathCrawler pour ces fonctions a une croissance assez lente (clairement sous-exponentielle) avec le paramètre `D`. Donc, comme présenti dans la Section 3, la génération de tests tous-les-chemins reste traitable pour ces programmes avec des centaines et des milliers de variables d'entrée, et PathCrawer fournit une méthode de génération efficace pour ces programmes.

D'autre part, la Figure 7 montre les résultats pour la fonction HC de la Figure 5a sur le graphe orienté complet G_N avec N nœuds (i.e. ayant un arc de i à j pour tous i, j). La colonne "chemins" montre le nombre de chemins faisables. Ici, en présence d'alias internes, PathCrawler génère des cas de tests superflus (*cf* [6]).

Comme l'avait prédit la Section 4, le temps de génération croît très rapidement (comme la factorielle environ), et la génération de tests tous-les-chemins devient intraitable déjà pour $N > 10$. Pour des graphes incomplets G, le nombre de tests et le temps de génération sont différents, mais leur croissance reste sur-exponentielle.

Nous avons essayé un exemple similaire issu d'un logiciel industriel avec plusieurs centaines de lignes de code C (qui ne sera pas décrit ici en détail pour des raisons de propriété intellectuelle), où les entrées ont été utilisées aussi comme des indices d'un tableau à deux dimensions. Nous avons obtenu des résultats similaires : la génération de tests tous-les-chemins devient déjà intraitable pour des programmes avec environ 20 variables d'entrée.

6 Conclusion et perspectives

On dit souvent que la génération de tests tous-les-chemins est intraitable sans vraiment donner une caractérisation des programmes pour lesquels elle peut être traitable, ou une description des structures de langage qui peuvent la rendre intraitable. Il semble important de pouvoir répondre à ces questions.

Cet article traite le problème d'évaluation de complexité potentielle de la génération de tests tous-les-chemins pour diverses classes de programmes. En utilisant le résultat de Pratt sur la résolution de contraintes de différence [10], nous avons prouvé un théorème qui stipule des conditions suffisantes pour qu'une classe de programmes permette une génération de test tous-les-chemins en temps polynomial. Il montre pour la première fois que la génération de test tous-

les-chemins peut être traitable pour certains programmes rencontrés en pratique.

Nous avons également construit une réduction originale du problème du circuit Hamiltonien pour montrer que la génération de tests tous-les-chemins pour une classe de programmes plus large, où les indices de tableaux et les décalages de pointeurs peuvent dépendre des entrées, est intraitable (NP-difficile). Cela donne un exemple concret des phénomènes de programmation qui peuvent rendre la génération de tests tous-les-chemins infaisable en pratique. Nous avons vu cette situation sur un exemple industriel. Ces résultats sont accompagnés de quelques expérimentations avec l'outil PathCrawler.

Les perspectives de travail incluent une étude détaillée des effets de divers phénomènes de langages de programmation sur les contraintes produites et la complexité de la génération de tests.

L'existence d'algorithmes polynomiaux de résolution pour d'autres types de contraintes (tels que "range contraintes") [11, 13] laisse espérer d'autres résultats positifs pour la génération automatique de tests qui permettront de mieux comprendre ses limites d'application en pratique.

Certes, le test exhaustif tous-les-chemins risque d'être intraitable dans de nombreux cas, mais la génération de tests pour d'autres critères de couverture (toutes-les-instructions, toutes-les-branches, etc.) peut être plus facile. Nous espérons que ce type d'études aidera les ingénieurs validation à anticiper la calculabilité et la complexité de la génération de tests et à choisir un critère de couverture approprié pour un programme donné.

Remerciements. Merci à Sébastien Bardin, Bernard Botella, Mickaël Delahaye, Philippe Herrmann, Bruno Marre et Nicky Williams pour de fructueuses discussions.

Références

[1] Sébastien Bardin and Philippe Herrmann. Structural testing of executables. In *ICST'08*, pages 22–31, Lillehammer, Norway, April 2008.

[2] Bernard Botella, Mickaël Delahaye, Stéphane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs : Experience with PathCrawler. In *AST'09*, Vancouver, Canada, May 2009.

[3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE : automatically generating inputs of death. In *CCS'06*, pages 322–335, Alexandria, VA, USA, November 2006.

[4] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed automated random testing. In *PLDI'05*, pages 213–223, Chicago, IL, USA, June 2005.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[6] Nikolai Kosmatov. All-paths test generation for programs with internal aliases. In *ISSRE'08*, pages 147–156, Seattle, WA, USA, November 2008.

[7] Nikolai Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development : New Prospects*, chapter XI : Constraint-Based Techniques for Software Testing. Advances in Intelligent Information Technologies Book Series. IGI Global, 2009.

[8] Bruno Marre and Agnès Arnould. Test sequences generation from Lustre descriptions : GATeL. In *ASE'00*, pages 229–237, Grenoble, France, September 2000.

[9] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In *CC'02*, pages 213–228, Grenoble, France, April 2002.

[10] V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, Cambridge, Massachusetts, USA, September 1977.

[11] G. Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E. Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3) :261–275, 1999.

[12] K. Sen, D. Marinov, and G. Agha. CUTE : a concolic unit testing engine for C. In *ESEC/FSE'05*, pages 263–272, Lisbon, Portugal, September 2005.

[13] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theor. Comput. Sci.*, 345(1) :122–138, 2005.

[14] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *ASE'04*, pages 290–293, Linz, Austria, September 2004.

[15] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler : automatic generation of path tests by combining static and dynamic analysis. In *EDCC'05*, pages 281–292, Budapest, Hungary, April 2005.

[16] Zhongxing Xu and Jian Zhang. A test data generation tool for unit testing of C programs. In *QSIC'06*, pages 107–116, Beijing, China, October 2006.

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