Mémoire de stage de fin d'étude d'ÉPITA
Michaël CADILHAC, sous la direction de Fatiha ZAÏDI, maître de conférences LRI
Juin - Octobre 2007

Marches aléatoires guidées uniformes pour le test de conformité de composants imbriqués

Résumé : ce rapport détaillera l'application d'un principe probabiliste de génération uniforme de chemins dans le cadre du test de conformité de composants imbriqués. Nous nous attacherons à améliorer un algorithme existant, HIT-OR-JUMP, par l'ajout de cette méthode probabiliste.

Une version électronique est disponible sur `http://michael.cadilhac.name/rapport-csi.pdf`

Table des matières

Introduction

Ce document présente les travaux réalisés pendant le stage de fin du Master 2 Recherche LMFI[1] et des études à l'ÉPITA[2] effectué au LRI[3] de juin à octobre 2007.

La vérification automatique en informatique est un domaine en pleine expansion sur laquelle se penchent de plus en plus les personnes intéressées par l'approximation. Et pour cause, la vérification est rapidement arrivée à un point où ses premières méthodes ne permettaient plus de répondre à certaines questions, du fait de la taille des systèmes à étudier.

Le test de conformité se propose de vérifier qu'une implémentation, d'autant plus sujette à l'erreur humaine qu'elle a été conduite par un programmeur, se conforme à sa spécification. Sans approximation, cela se résume ou bien au problème d'isomorphisme de graphe, ou bien à la génération exhaustive de tous les cas d'utilisation d'un système ; impensable.

« Tout est approximation », comme disent les physiciens, et c'est à partir de ces considérations qu'est née la branche du test de conformité. Le principe fondamental du test est l'utilisation de méthodes d'extraction de cas d'utilisation *pertinents* du système, puis la mise à l'épreuve de l'implémentation avec ce cas.

Le présent travail se focalise sur une branche du test qui s'intéresse à vérifier qu'un composant parmi tout un système est bien implémenté. Nous présentons dans ce rapport dans un premier temps le contexte du stage, et la vie en entreprise. Nous passons alors à la partie technique développant le problème. Dans celle-ci, nous abordons quelques concepts généraux de test en informatique, puis étudions un algorithme qui constitue une base de ce document, HIT-OR-JUMP. Nous voyons ensuite comment il est possible de naviguer sur un graphe que l'on ne construit pas en s'assurant d'un critère d'uniformité, puis mélangeons HIT-OR-JUMP et ce dernier ingrédient. Enfin, nous faisons une incursion dans le futur de ce travail, et en particulier, dans la fin de ce stage.

[1] LMFI : `http ://www.logique.jussieu.fr/www.dea/`
[2] ÉPITA : `http ://www.epita.fr`
[3] LRI : `http ://www.lri.fr`

CHAPITRE

1 Cadre du stage

1.1 Le LRI

Le laboratoire de Recherche en Informatique (LRI), créé il y a plus de 30 ans à l'Université Paris-Sud, est l'un des plus grands et des plus prestigieux laboratoires français de recherche en informatique. Au 1^{er} janvier 2004, le laboratoire comptait 177 membres : 78 chercheurs et enseignants-chercheurs permanents, 63 doctorants, 22 personnels non permanents et 14 personnels techniques et administratifs.

Le LRI est constitué de 10 équipes de recherche, assistées d'une équipe de support système et réseau et d'une équipe administrative. Le laboratoire occupe 4500m2 dans le bâtiment 490 sur le campus d'Orsay. Ses principaux moyens de recherche sont un réseau haute performance, une grappe de calcul et une bibliothèque.

Le LRI est une unité mixte de recherche (UMR 8623) du CNRS et de l'Université ParisSud. Les personnels permanents sont issus de ces deux organismes (14 chercheurs et 10 ITAs CNRS, 64 enseignants-chercheurs et 4 IATOS Université Paris-Sud). Le budget annuel, hors salaires des permanents, est de l'ordre de 1.6M. Moins d'un tiers de ce budget provient des dotations de base du CNRS et de l'Université ; le reste provient de contrats et subventions obtenus par les membres du laboratoire.

Les thèmes de recherche abordés par le LRI couvrent un large spectre de l'informatique : algorithmique, complexité, calcul quantique, théorie des graphes, fondements des communications, micro-architecture, clusters et grilles de calcul, génie logiciel, programmation, interaction homme-machine, bases de données, systèmes d'inférence, fouille de données, apprentissage par machine, et bioinformatique.

Cette diversité est l'une des forces du laboratoire, car elle favorise les recherches aux frontières des thématiques, là où le potentiel d'innovation est le plus grand. Ainsi, il y a deux ans, des chercheurs de quatre équipes ont décidé de créer un axe transversal sur la bioinformatique. Cet axe est désormais une équipe à part entière qui développe une approche originale en combinant les concepts de différents domaines, comme l'apprentissage par machine et les algorithmes randomisés pour prédire l'information biologique pertinente dans les données génomiques.

Le LRI est lui-même subdivisé en 10 équipes selon des orientations de recherche différentes :

– Théorie des graphes et fondements des communications,

– Algorithmique et complexité,
– Programmation,
– Bases de données,
– Preuves et programmes,
– Architectures parallèles.
– Parallélisme,
– Intelligence artificielle et systèmes d'inférence,
– Inférence et apprentissage,
– Bioinformatique.

L'organigramme suivant précise l'intégration des différentes équipes au sein même du LRI.

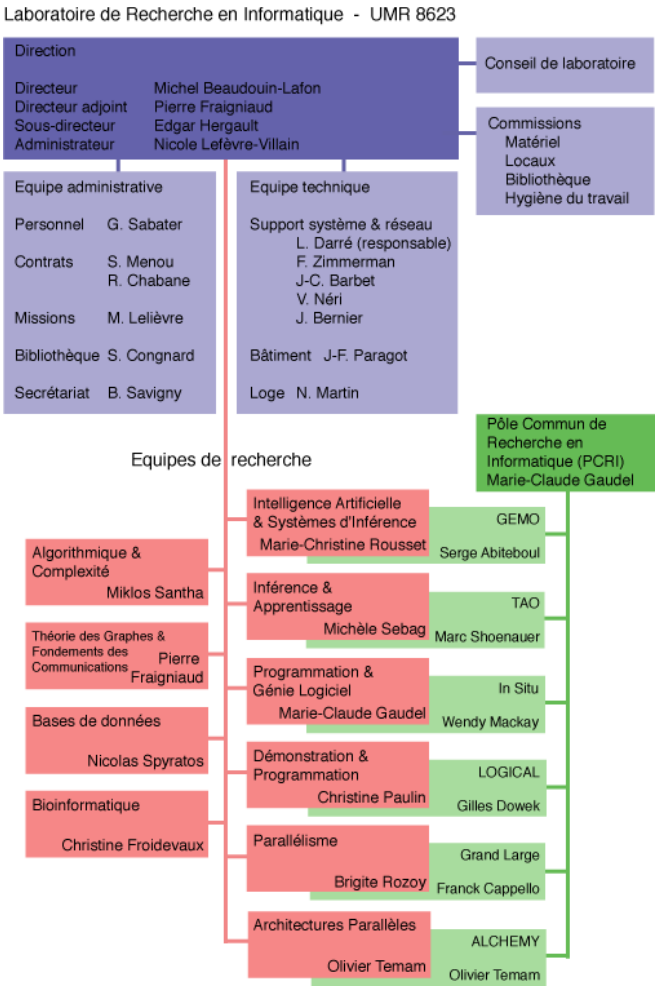

FIG. 1.1: Organisation du LRI

1.2 Le service et l'équipe

Le travail effectué prenait place dans l'équipe nouvellement appelée ForTesSE[1], pour *Formal Testing and System Exploration*, dirigée par Marie-Claude GAUDEL, directrice de recherche au LRI. J'étais aiguillé dans toutes mes démarches par mon maître de stage, le docteur Fatiha ZAÏDI.

Cette très récente équipe se focalise principalement sur le test formel, comprendre la vérification qu'un système donné se comporte bien vis-à-vis de certaines propriétés requises. C'est dans cette branche toute particulièrement que mon travail s'est inscrit.

Pour autant, le deuxième sujet de prédilection de l'équipe ne fut pas en reste ; en effet, la problématique d'exploration de système était, pour ainsi dire, le véritable prétexte de mon étude : il s'agissait en effet de rassembler deux résultats indépendants de ces deux domaines.

L'exploration de système pose une grande quantité de questions. Par système, il est généralement entendu « système de modules ». Ces modules étant des graphes, nous pouvons, bien évidemment, calculer le système entier pour l'explorer. Le problème principal est celui de l'explosion combinatoire du nombre d'états du système résultat. Il est en effet souvent impossible de construire un objet aussi gros.

La solution proposée généralement est alors de parcourir les modules séparément. Mais les questions qui découlent de cette possibilité sont nombreuses, venant principalement du fait que la connaissance nécessaire de certaines propriétés du système global ne se trouve plus aussi facilement.

[1] ForTesSE : `http ://www.lri.fr/~oudinet/fortesse/pubs/fortesse.html`

CHAPITRE

2 Contexte et rappels

Ce chapitre présente les notions et le vocabulaire essentiels à la compréhension du reste du document. Le lecteur familier avec le domaine du test de composants et les machines à états communiquantes pourra sans dommage passer rapidement cette partie.

Tout d'abord, intéressons-nous à légitimer l'apport de pareilles notions. En effet, en toute généralité le présent document et les travaux entrepris durant cette étude pourraient s'appliquer à un certain type de parcours aléatoire dans des graphes, et c'est à une fin double que l'on suivra l'étude au travers du test de composants. Tout d'abord parce que cela demandera un effort d'imagination moindre chez le lecteur, qui pourra ainsi se concentrer sur les résultats ; ensuite parce qu'il est toujours intéressant de pouvoir travailler sur une implémentation[1] réelle pour avoir des résultats empiriques.

Le lecteur soucieux de connaître les résultats dans un contexte plus général pourra se reporter à l'ouverture faite au chapitre 5.

Ce chapitre donne dans un premier temps une vue d'ensemble du test en informatique, puis se focalise sur le test imbriqué. Pour cela sont abordées, en plus des notions générales, quelques considérations sur la formalisation de spécifications et les méthodes de test classiques.

Sauf mention contraire, toutes les définitions sont extraites de [CLRZ99] [LY96] [LSKP96] et [GM03].

2.1 Le test en informatique

La norme IEEE 829 donne la définition suivante du test :

Définition 2.1.1 *Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification.*

Plus généralement, le test cherche à montrer la présence de différences entre une implémentation et une définition formelle des fonctionnalités requises.

[1] Le verbe « implémenter » et ses dérivés sont autorisés dans la langue française par l'arrêté du 27 juin 1989 relatif à l'enrichissement du vocabulaire de l'informatique (Journal Officiel du 16 septembre 1989).

D'autres branches de la vérification, comme le *model-checking* [CGP00] ou le test de propriétés [Gol97] s'intéressent principalement à vérifier que le système donné répond à certaines attentes arbitraires (atteinte d'un état, propriété de bouclage, etc) en ne se concentrant pas forcément sur la conformité vis-à-vis d'une spécification exhaustive.

Nous nous focaliserons sur une branche de test en boîte noire, c'est-à-dire avec un système dont on ne peut pas examiner les états internes. Plus précisément, il s'agira de vérifier le bon comportement d'une partie d'un système sans pouvoir isoler ledit composant dans l'implémentation.

2.2 Test de composants imbriqués

Posons tout d'abord les premières notions et le vocabulaire :

Définition 2.2.1 *On appellera* composant *ou* module *une entité indivisible qui dispose d'*entrées *et de* sorties.

Les composants s'assemblent pour créer un système *dans lequel ils communiquent entre eux aussi bien :*

- *De manière synchrone par rendez-vous, c'est-à-dire sans modélisation d'un canal de communication,*
- *De manière asynchrone, c'est-à-dire en considérant l'existence de tampons (*buffer*), tels que l'on en retrouve, par exemple, en réseau.*

On remarque que le premier type de communication peut aisément simuler le second, par l'ajout de composants simulant les tampons, nous ne considérerons de fait que des communications synchrones en nous autorisant tout de même à parler de communications asynchrones.

On appellera environnement *tout ce qui est extérieur au système ; ce sont les interactions du systèmes avec l'environnement qui seront effectivement observables.*

En effet, le système implémenté *est lui-même considéré comme une boîte noire ; on suppose que l'on sait qu'il y a les composants, mais qu'on ne peut pas les isoler.*

Nous aurons donc d'une part un système implémenté, on dira simplement « système », et les spécifications des divers modules, sous une forme que nous établirons plus loin.

Dans ce cadre, notre but est de vérifier qu'une partie d'un système donné, un composant, se conforme à une *spécification* ; en d'autres termes, qu'une partie de l'implémentation fonctionne correctement. C'est à ce titre que le présent domaine s'appelle le *test de conformité de composants imbriqués.*

Prenons un exemple. Soit le système suivant :

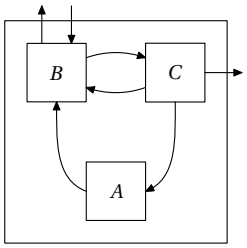

FIG. 2.1: Système α

Ce système est formé à partir des trois composants A, B et C, qui communiquent entre eux de manière asynchrone, les tampons étant symbolisés par les flèches entre eux. Notre objectif sera de vérifier qu'un des composants se comporte bien *dans le contexte* des deux autres composants.

Ainsi, si l'on nomme A_{spec} la spécification de A, on veut vérifier que $A_{\text{spec}} \equiv A$ *dans le contexte* des composants B et C, ce que l'on notera $\{B,C\} \times A_{\text{spec}} \equiv \{B,C\} \times A$. L'utilisation la graphie $\cdot \times \cdot$ sera amplement légitimée par la suite.

Il est à remarquer que le contexte est d'une importance cruciale pour la vérification ; en effet, vérifier qu'un module implémenté se comporte comme sa spécification sur toutes les entrées, fussent-elles complètement inattendues est non seulement bien plus fastidieux mais aussi complètement insensé. Il faut en effet se cantonner à *ce qui peut arriver* et c'est le contexte qui met ces conditions en place. De plus, il faut garder en mémoire le fait que dans l'implémentation, le contexte et le composant sous test sont indissociables.

2.3 Implémentation et spécification

L'implémentation n'étant, en définitive, qu'une boîte noire, nous n'avons presque aucune restriction à faire dessus. En fait, qu'elle soit faite en C, en assembleur ou à base d'automates, ce qui importe principalement est que l'on puisse observer ses sorties sur une entrée donnée. Nous supposerons aussi, comme il est classique de le faire, que l'on peut poser une borne supérieure à son nombre d'états, de sorte qu'il ne soit pas possible que l'implémentation contienne de manière exhaustive les réponses à toutes les séquences de test possibles.

Il y a beaucoup plus à dire sur la spécification. On se rappelle que nous avons besoin d'une spécification pour chaque module et que ces spécifications doivent « discuter » entre elles. Il est essentiel pour la suite que l'on puisse observer ces spécifications *étape par étape*, et que l'on sache à chaque branchement quelle est la portion de la donnée de l'environnement qui a fait choisir tel ou tel chemin.

Le traitement de donnée en entrée/sortie et la connaissance parfaite des états amène naturellement à considérer des transducteurs, les automates avec sortie. Les besoins en communications nous amènent à considérer des automates *communicants*, et pour se placer dans un cas plus général, nous opterons pour des automates *étendus*.

Définition 2.3.1 (Automate étendu communiquant (ECFSM)) *Un automate étendu communiquant est un quintuple $\mathscr{A} = \langle I, O, S, \vec{x}, T\rangle$ où :*

- *I et O sont les ensembles finis de symboles d'entrée et de sortie,*
- *S est l'ensemble fini des états,*
- *$\vec{x}$ est l'ensemble des variables de l'automate,*
- *T est l'ensemble des transitions.*

Chaque transition est de la forme $\tau = \langle s, q, i, o, P, A\rangle$ où :

- *s et q sont les états de départ et d'arrivée de τ,*
- *i et o l'entrée et sortie de la transition,*
- *P est un prédicat sur $\vec{x}$ qui indique si la transition peut être franchie,*
- *A est une affectation sur $\vec{x}$ de manière à ce que $\vec{x} := A(\vec{x})$ après franchissement de la transition.*

L'automate ainsi défini est dit déterministe *si d'un état ne part au maximum qu'une transition avec une étiquette donnée. Dans le cas contraire, il est dit* nondéterministe.

Certaines transitions, dites de communication, sont étiquetées en entrée par un élément de l'ensemble des primitives de synchronisation $C \subseteq I$. Pour franchir une pareille transition, tous

les autres composants qui ont une transition étiquetée par cette même entrée doivent passer une transition de ce type au même moment.

Dans l'absolu, le choix de ce formalisme est tout à fait arbitraire et nous aurions pu considérer aussi bien des systèmes de transitions (étiquetés) ou des réseaux de Pétri (cf. [Mil95] [Pet62] [Bri88]). La présente étude pourrait être abstraite à toute description pouvant être représentée sous forme de graphe.

Reprenons l'exemple figure 2.1. On comprend bien que, d'un point de vue spécification, on se retrouve avec trois automates $\mathscr{A}_{\text{spec}}$, $\mathscr{B}_{\text{spec}}$ et $\mathscr{C}_{\text{spec}}$. Le système spécification global est l'automate combinaison de toutes ces spécifications, avec toutes les transitions intervenant à n'importe quel moment. Ce système dans lequel à un instant t un des trois automates « avance » est très précisément le produit asynchrone :

Définition 2.3.2 (Produit asynchrone de ECFSM) *Soient deux ECFSM $\mathscr{A}_i = \langle I_i, O_i, S_i, \vec{x}_i, T_i \rangle$ pour $i = \{1,2\}$ tels qu'ils soient dissociés, c'est-à-dire que $S_1 \cap S_2 = \emptyset$ et $\vec{x}_1 \cap \vec{x}_2 = \emptyset$.*

Le produit asynchrone d'automate *de $\mathscr{A}_1$ et $\mathscr{A}_2$, noté $\mathscr{A}_1 \times \mathscr{A}_2$ est l'automate $\mathscr{A}_{12} = \langle I, O, S, \vec{x}, T \rangle$ tel que :*

- *$I = I_1 \times I_2$ et $O = O_1 \times O_2$, où $\cdot \times \cdot$ est le produit cartésien,*
- *$S = S_1 \times S_2$,*
- *$\vec{x} = \vec{x}_1 \times \vec{x}_2$.*

Pour ce qui est de la liste des transitions T, notre construction se fera en considérant toutes les transitions des deux automates. Ainsi, soient

$$\tau_i \in T_i, \tau_i = \langle s_i, q_i, i_i, o_i, P_i, A_i \rangle, i \in \{1,2\}.$$

Deux types de transitions sont alors présents dans l'automate produit :

- *D'une part, pour tout nœud p_2 de $\mathscr{A}_2$, il y a une transition dans $\mathscr{A}$ constituée par :*
  - *Son état de départ (s_1, p_2) et d'arrivée (q_1, p_2),*
  - *Son entrée (i_1, ε) avec ε le mot vide, et sa sortie (o_1, ε),*
  - *Son prédicat $P_1(\pi^1(\vec{x}))$, avec $\pi^j(\vec{x})$ la projection sur la j-ième composante de $\vec{x}$,*
  - *Son assignation $\vec{x} := A_1(\pi^1(\vec{x})) \times \pi^2(\vec{x})$.*

  L'automate produit contient aussi les transitions issues de la construction symétrique pour τ_2 et tout nœud de $\mathscr{A}_1$.
- *D'autre part, pour chaque paire de transitions précédemment définies, il y a une transition dans $\mathscr{A}$ constituée par :*
  - *Son état de départ (s_1, s_2) et d'arrivée (q_1, q_2),*
  - *Son entrée (i_1, i_2) et sa sortie (o_1, o_2),*
  - *Son prédicat $P_1(\pi^1(\vec{x})) \wedge P_2(\pi^2(\vec{x}))$,*
  - *Son assignation $\vec{x} := A_1(\pi^1(\vec{x})) \times A_2(\pi^2(\vec{x}))$.*

Nous n'entrerons pas dans les considérations qui voudraient que l'on distingue les états stables (globalement observables) et les états de passage (provenant de transitions sans communication des automates spécifications). On suppose donc notre automate résultant entièrement stable et observable. Pour plus de détail sur les complications que cela entraîne, principalement liées au fait que l'on ne sache une transition franchie qu'avec une certaine probabilité, se référer à [LSKP96].

Il se remarque donc rapidement que le produit asynchrone de plusieurs modules entraîne une explosion combinatoire du nombre d'état de l'automate résultant. C'est ici un des points

les plus importants de la motivation des méthodes décrites plus loin : on ne veut pas construire l'automate spécification du système en entier, car ce serait trop coûteux.

La proposition suivante donne un indice de la difficulté de la tâche :

Proposition 2.3.3 ([LSKP96]) *Les problèmes suivants sont PSPACE-complets :*
- *Savoir si un état ou une transition d'un composant est atteint à partir de l'état initial du système spécifié,*
- *Savoir si un état ou une transition de l'automate combiné est atteint à partir de l'état initial de ce dernier.*

2.4 Méthodes de test

Les méthodes présentées ici visent à générer des entrées « efficaces » dont on calcule une sortie attendue et à vérifier sur l'implémentation que le résultat est correct. Des entrées « efficaces » sont des entrées qui couvrent au maximum les états ou les transitions du module ou de la fonctionnalité à tester.

Générer de tels chemins est une chose relativement simple lorsque l'on dispose de la spécification en tant que produit d'automates et que l'on sait d'où chaque transition provient (de quel module avant le produit). Ces méthodes de génération de tests, dites *structurées* (e.g., [Yan91]) sont, comme nous l'avons évoqué, trop coûteuses pour nos besoins. L'autre extrême de la génération de tests, non abordé ici, est le *test passif*, où il n'y a aucun contrôle sur la séquence de test (e.g., [Sei72]).

Élagage et raffinement

Une première méthode, proposée dans [LSKP96] dans le cadre de la vérification d'implémentation de protocoles, s'intéresse tout particulièrement aux fonctionnalités d'une spécification, au sens de l'utilisateur final. Elle est inspirée des méthodes de projection de protocoles de [LS82] où l'on analyse des protocoles restreints en lieu et place du protocole original.

Il s'agit dans un premier temps d'identifier et de décrire les *services*, au sens intuitif, que propose le protocole et de les modéliser sous forme d'automates. Ces services utiliseront des *primitives de services* comme opérations d'entrée/sortie, qui agissent comme des communications de haut niveau. Ici, c'est un service que l'on cherche à vérifier, bien que l'on dispose, comme précédemment, des spécifications des composants.

À partir de ce service et de chacune des spécifications :

(i) On supprime dans les spécifications les transitions qui font intervenir des primitives de services autres que celles du service considéré,

(ii) S'il y a des primitives d'entrée qui n'ont plus de contrepartie de sortie, on supprime les transitions concernées, et inversement,

(iii) Enfin, on récupère les composantes fortement connexes partant des états initiaux de chaque automate spécification, puis on retourne en (ii) s'il y a lieu.

L'idée intuitive derrière cet algorithme est que l'on peut supprimer les parties du protocole qui ne peuvent être atteintes que par application de primitives d'entrée (resp. sortie) sans primitives de sortie (resp. entrée) correspondantes, et que les automates résultants doivent être fortement connexes.

L'opération permet d'obtenir des automates spécifications élagués, leur produit pouvant être ainsi construit de manière plus efficace. L'utilité de cette méthode réside dans la proposition suivante :

Proposition 2.4.1 ([LSKP96]) *Une faute dans le système implémenté détectée par un test généré à partir du produit d'automates élagués est une erreur dans l'implémentation vis-à-vis de la spécification.*

On remarque, pour autant, que la réciproque n'est pas forcément vraie. En effet, nous nous sommes juste assuré que l'ensemble d'automate ainsi élagué fournissait le service concerné, mais les interactions avec les autres services ont été réduites, alors qu'elles auraient pu causer des fautes dans l'implémentation.

Marche aléatoire

S'il est difficile de connaître la structure entière de l'automate composite, il est possible de le construire localement, en parcourant tous les automates en même temps sans construire le produit. Il s'agit donc de considérer l'état courant du système comme un vecteur des états des composants.

Le bien-fondé de cette méthode est donné par la proposition suivante :

Proposition 2.4.2 ([LSKP96]) *Si une séquence de test est un test de conformité sur chacune des spécifications des composants, elle l'est aussi pour l'automate produit entier.*

On transpose aisément ce résultat à notre cas de vérification d'un module avec les autres constituant le contexte. La marche aléatoire non biaisée suivante est la plus simple à mettre en œuvre :

Algorithme 1 Marche aléatoire non biaisée [CLRZ99]

ENTRÉES: Les composants $\mathscr{A}_i, i \in [\![1..r]\!]$ dont $\mathscr{A}_k$ à tester
SORTIES: Une séquence de test
  $e \leftarrow$ états initiaux des automates
  $\sigma \leftarrow$ ""
  tant que toutes les transitions de $\mathscr{A}$ ne sont pas visitées **faire**
    (i) $e \leftarrow$ successeur aléatoire de manière uniforme dans le produit
    (ii) Marquer la transition empruntée
    (iii) $\sigma \leftarrow \sigma +$ étiquettes d'entrée des transitions empruntées
  fin tant que
  Retourner σ

Il n'est donc bien évidemment pas nécessaire de construire le produit complet, mais seulement d'avoir une connaissance locale de la position dans les automates et de se souvenir du chemin parcouru.

Un des principaux griefs que l'on peut avoir contre cette méthode est illustré par l'exemple suivant :

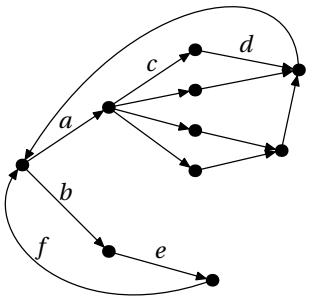

FIG. 2.2: Topologie problématique pour la marche aléatoire

Une marche aléatoire de taille de chemin 3 dans ce système renverra le chemin a, c, d avec une probabilité de $0,5 \times 0,25 \times 0,25 = 0,03125$, tandis que le chemin b, e, f a une probabilité $0,5$ d'apparaître ; le problème est que tous les chemins n'ont pas une probabilité d'apparition égale, il n'y a pas distribution uniforme sur les chemins de taille 3 avec une marche aléatoire.

Marche aléatoire guidée

Une étape d'évolution naturelle après cette apologie de l'aléa est d'orienter le choix du successeur pour obtenir des résultats potentiellement meilleurs. De différents critères d'uniformité viendront naturellement comme solutions à cette problématique.

La méthode élémentaire derrière une marche aléatoire guidée est donc de donner la priorité aux transitions du composant testé. Plus précisément, il s'agit dans l'étape *(i)* de l'algorithme 1 de privilégier les transitions du module qui n'ont jamais été parcourues, et si on n'en a pas la possibilité, les transitions du module déjà parcourues. En dernier ressort, on emprunte une transition ne provenant pas du module testé.

La marche aléatoire guidée requiert donc une mémoire équivalente à la marche aléatoire non biaisée, c'est-à-dire en $\mathcal{O}(|T|)$ avec T les transitions du module testé. Notons que le critère d'uniformité n'est pas ici privilégié : le choix entre les transitions se fait certes dans trois ensembles distincts, mais une fois un groupe choisi, elles sont prises aléatoirement de manière uniforme dans ce dernier.

Remarquons enfin que les marches aléatoires peuvent conduire à des résultats peu intéressants sur certaines topologies où l'algorithme reste « bloqué » dans une petite partie du graphe, en tournant en rond. Nous présenterons en section 3.1 une solution à ce problème.

CHAPITRE

3 Fondements de l'étude

La présente étude se consacre dans un premier temps à l'intégration d'une méthode probabiliste de tirage uniforme de chemins dans une procédure de génération de tests. Ce chapitre présente les deux ingrédients majeurs de ce travail ; d'une part l'algorithme HIT-OR-JUMP, qui est un générateur de tests formé d'une combinaison paramétrable d'une méthode structurée et d'une marche aléatoire guidée, et d'autre part une technique pour effectuer des tirages uniformes de chemins dans de grands modèles.

3.1 Génération de tests avec HIT-OR-JUMP

Motivations

La création de l'algorithme HIT-OR-JUMP [CLRZ99] puise ses motivations dans les problèmes que rencontrent les marches aléatoires présentées en section 2.4. Plus précisément, par l'introduction d'une saveur structurée, l'algorithme se propose de pallier les trois défauts suivants :

1. La marche peut tourner en rond, comme évoqué en section 2.4,
2. Le passage de goulots d'étranglement dans le graphe se fait difficilement, ne testant alors pas ce qu'il y a après,
3. Même si une transition du module testé est « proche », elle peut ne pas être considérée.

Attardons-nous quelque peu sur ces points.

Ce que l'on appelle goulot d'étranglement est illustré dans la figure suivante, où l'on se place, pour éclaircir le dessin, dans une partie du graphe sans arcs en arrière :

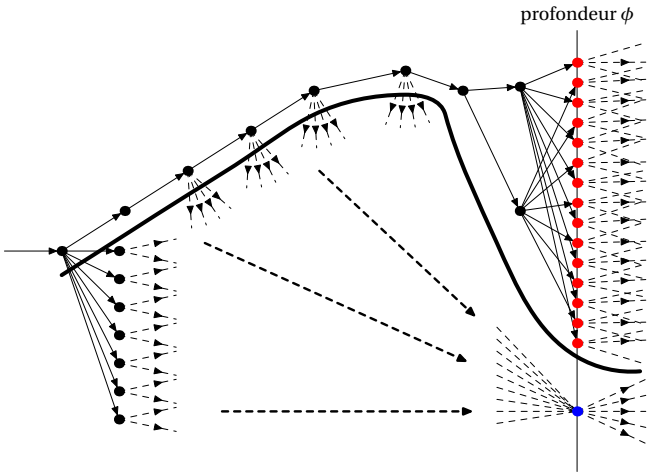

FIG. 3.1: Goulot d'étranglement

Nous voyons donc qu'un parcours de type marche aléatoire (section 2.4) d'une profondeur ϕ aura très peu de chance d'atteindre un des points rouges. En effet, la plupart des parcours atteindront le point bleu, qui a une probabilité d'apparition bien plus grande.

Le troisième problème met en avant les faiblesses de la marche aléatoire guidée : si elle est en effet capable de s'orienter de manière à privilégier certaines transitions, elle ne peut le faire que sur les successeurs du nœud courant. Il est au final dommage de ne pas pouvoir s'orienter de manière à atteindre une transition intéressante si cette dernière est un nœud plus loin. Autrement dit, il n'y a pas de *look-ahead*, de petites incursions en avant.

Algorithme

Le fonctionnement d'HIT-OR-JUMP repose sur le principe suivant. Après k itérations, on se retrouve sur un nœud, mettons s_k. Un parcours *exhaustif* d'une profondeur bornée donnée est alors fait. Si durant ce parcours une transition non parcourue du module à tester est trouvée, le chemin vers le point destination s_{k+1} de cette transition est sauvé – pour recréer la séquence de test – et l'on reprend à partir de s_{k+1}. Dans le cas contraire, on repart d'une feuille quelconque de l'arbre de parcours.

Pour le vocabulaire, il y a un HIT si l'on trouve une transition non parcourue dans l'arbre de parcours, sinon il s'agit d'un JUMP.

L'algorithme est le suivant :

Algorithme 2 HIT-OR-JUMP [CLRZ99]

ENTRÉES: Les composants $\mathscr{A}_i, i \in [\![1..r]\!]$ dont $\mathscr{A}_k$ à tester, et $\phi \in \mathbb{N}^*$
SORTIES: Une séquence de test
  $s \leftarrow$ vecteur des états initiaux des automates
  $\sigma \leftarrow$ ""
  tant que toutes les transitions de $\mathscr{A}_k$ ne sont pas visitées **faire**
    À partir de s, faire un parcours (largeur ou profondeur) borné en profondeur par ϕ dans le produit
    si on atteint une transition τ contenant une transition non marquée de $\mathscr{A}_k$ **alors**
      {Partie HIT }
      $\sigma \leftarrow \sigma$ + entrées des transitions de s à τ
      Marquer la transition empruntée de $\mathscr{A}_k$
      $s \leftarrow$ destination de la transition τ
    sinon
      {Pas de transition non marquée à une distance de ϕ}
      {Partie JUMP }
      Choisir aléatoirement de manière uniforme une feuille l de l'arbre de recherche
      $\sigma \leftarrow \sigma$ + entrées des transitions de s à l
      $s \leftarrow l$
    fin si
  fin tant que
  Retourner σ

Cette version épurée de l'algorithme peut être modifiée pour choisir, par exemple, dans la partie JUMP, une feuille selon une priorité donnée à ses transitions de sortie, comme nous le faisions section 2.4. Au même titre que nous avions la marche aléatoire *guidée*, nous aurions le JUMP *guidé*.

Voyons maintenant le comportement de l'algorithme selon la valeur de ϕ :

- Si $\phi = 1$, l'algorithme ne regarde que les transitions sortant du nœud courant. Il choisit ensuite en priorité (ce serait un HIT) une transition non marquée, et sinon (un JUMP) prend une transition aléatoirement. Il s'agit précisément d'une marche aléatoire guidée,
- Si $\phi = \infty$, dans le pire des cas, on va parcourir, en se souvenant des chemins comme le nécessite la nature du parcours, l'intégralité de l'automate produit. On aura au final construit le produit, ce qui nous ramène à une recherche structurée.

Enfin, reprenons les défauts que voulait corriger l'algorithme.

1. Le blocage est solutionné partiellement grâce à la possibilité de faire un JUMP,
2. Le passage de goulots d'étranglement est simplifié du fait que le choix aléatoire se fasse sur les *feuilles* de l'arbre de parcours (dans la figure 3.1, cela correspond à la barre en ϕ),
3. Les transitions « proches » sont atteintes grâce au parcours du HIT ; l'algorithme reproduit en effet le principe du *look-ahead* par ce parcours exhaustif.

3.2 Génération uniforme de chemins

Nous présentons dans cette section les résultats de [DGG+06] sur la génération uniforme de chemins dans de grands modèles, qui ne requièrent pas la construction de l'automate produit.

L'idée de la méthode réside dans le fait que l'on sache extraire de manière efficace un chemin uniformément sur chacun des automates descriptions, et qu'en combinant astucieusement des chemins de certaines tailles pris dans chaque automate on obtient un « chemin uniforme » sur le produit.

Nous voyons donc dans cette section comment générer de manière uniforme des chemins dans un automate entièrement connu, puis étudions quelles « parts » doivent jouer chaque automate dans le chemin global et comment combiner les chemins de chaque automate. Enfin, nous abordons quelques considérations relatives aux communications par synchronisations vu dans la définition 2.2.1.

Génération uniforme de chemins dans un automate

Nous voyons dans un premier temps comment générer uniformément un chemin dans un automate en une complexité polynomiale en son nombre d'états. Il est donc bien clair que cette méthode ne sera pas utilisée sur le produit mais, comme dit en introduction, sur chacun des composants.

La méthode proposée ici, entrevue dans [HC83] et approfondie dans [FZC94], pour la génération uniforme de chemins de taille donnée est une marche aléatoire guidée dans l'automate. Lorsque l'on doit sélectionner le successeur, il est ici choisi en fonction de probabilités préalablement calculées.

Tout d'abord, posons pour tout nœud s de l'automate, $g_m(s)$ le nombre de chemins partant de s de taille m. Mettons nous à un instant quelconque de la génération : nous sommes sur un état s de successeurs $(s_i)_{i\in[\![1..k]\!]}$, et il reste m transitions pour terminer notre chemin de taille n. La condition pour l'uniformité est alors clairement que s_i doit être choisi avec probabilité $g_{m-1}(s_i)/g_m(s)$.

On doit donc calculer la valeur de $g_i(s)$ pour tout s et pour $i \in [\![0..n]\!]$. Il suffit de poser la relation de récurrence suivante :

$$\begin{aligned} g_0(s) &= 1 \\ g_i(s) &= \sum_{s' \text{ successeur de } s} g_{i-1}(s') \text{ pour } i \in [\![1..n]\!] \end{aligned} \tag{3.1}$$

On a donc besoin pour réaliser l'opération complète, d'une mémoire en $\mathcal{O}(n \times |Q|)$ avec Q les états de l'automate pour stocker les g_i, et d'un temps en $\mathcal{O}(n \times |Q|)$ pour les deux opérations de calcul et de génération.

Des composants au produit

De plus, les chemins sur l'automate composite peuvent être vus comme un enchevêtrement de chemins sur les automates composants ; au même titre que le produit est l'enchevêtrement poussé à son paroxysme des composants eux-mêmes. Nous appellerons donc un mélange de chemins le chemin sur le produit obtenu par le procédé suivant :

Algorithme 3 Mélange uniforme de chemins [DGG+06]

ENTRÉES: r chemins $w_1, \ldots, w_r$ de tailles resp. $n_1, \ldots, n_r$
SORTIES: Un chemin w de taille $\sum n_i$
  $w \leftarrow$ ""
  $n \leftarrow \sum n_i$
  tant que $n > 0$ **faire**
    Choisir un i entre 1 et r avec une probabilité $\frac{n_i}{n}$
    Ajouter la première étape de w_i à la fin de w
    Supprimer la première étape de w_i
    $n_i \leftarrow n_i - 1$
    $n \leftarrow n - 1$
  fin tant que
  Retourner w

Chaque chemin extrait uniformément à partir des composants aura une taille dépendante du *poids* de sa spécification. Dans une génération uniforme, il est clair que ce poids est tout simplement le nombre de chemins de taille n que peut générer le composant. On a vu précédemment que ce calcul était acceptable, avec une complexité abordable.

Il ne reste plus qu'à trouver quelles sont les tailles exactes $n_1, \ldots, n_r$ à utiliser pour qu'une fois mélangé on ait un « chemin uniforme » sur le produit. C'est ce qu'avance cette proposition :

Proposition 3.2.1 ([DGG+06]) *Soient $\mathscr{A}_i, i \in [\![1..r]\!]$ les automates spécifications et $l_i(n)$ le nombre de chemins de taille n dans l'automate $\mathscr{A}_i$. Soit enfin $l(n)$ le nombre de chemins de taille n dans l'automate composite. Pour obtenir un tirage uniforme des chemins après mélange des chemins $(w_i)_{i\in[\![1..r]\!]}$ des composants, il suffit que les tailles des chemins $(n_i)_{i\in[\![1..r]\!]}$ soient tirées avec une probabilité*

$$\boldsymbol{Pr}(n_1, \ldots, n_r) = \frac{\binom{n}{n_1, \ldots, n_r} l_1(n_1) l_2(n_2) \ldots l_r(n_r)}{l(n)}$$

La proposition 3.2.1 requiert la connaissance d'un objet complexe, $l(n)$. En effet, avec les notations précédentes et si l'on suppose connus les $l_i(n)$, il est clair que

$$l(n) = \sum_{k_1+\cdots+k_r=n} \binom{n}{k_1, k_2, \ldots, k_r} l_1(k_1) l_2(k_2) \ldots l_r(k_r) \tag{3.2}$$

Or si nous avions échappé à l'explosion combinatoire en ne prenant pas le produit entier, nous avons ici quelque chose de très semblable et d'aussi coûteux, en terme de complexité, à calculer. Il nous faut donc approximer la valeur de cet objet là.

Pour cela, rappelons le théorème suivant :

Théorème 3.2.2 ([FS06]) *Soient $\mathscr{A}$ un automate et $l(n)$ le nombre de chemins dans $\mathscr{A}$ de longueur n. Il existe un entier N_1, un ensemble fini de nombres complexes $\omega_1, \ldots, \omega_k$ et un nombre fini de polynômes $R_1(n), \ldots, R_k(n)$ tels que :*

$$n \geq N_1 \rightarrow l(n) = \sum_{j=1}^{k} R_j(n) \omega_j^n.$$

Chacun de ces objets peut être calculé en temps polynomial en la taille de l'automate $\mathscr{A}$.

De plus, si l'automate $\mathscr{A}$ est apériodique et fortement connexe alors il y a un unique i tel que $|\omega_i| > |\omega_j|$ pour $i \neq j$. On a alors que pour tout n, $R_i(n) = C$ avec C une constante, et donc, en posant $\omega = \omega_i$, on a asymptotiquement :

$$l(n) \sim C\omega^n \text{ et } \frac{C\omega^n}{l(n)} \text{ converge vers 1 exponentiellement vite.}$$

Remarquons tout d'abord que dans notre cas, les automates descriptions n'ont que des états finals, ce qui satisfait l'apériodicité. Enfin, nous pouvons toujours considérer nos automates fortement connexes puisqu'ils disposent généralement d'un *reset*.

Le résultat fort connu du théorème 3.2.2 n'est bien évidemment pas à appliquer directement à $l(n)$, puisque cela nécessiterait un temps de calcul polynomial, certes, mais en la taille du produit. Prenons plutôt une approximation de chaque $l_i(n)$, en posant :

$$l_i(n) \sim C_i\omega_i^n \text{ pour } i \in [\![1..r]\!].$$

On a ensuite, en reprenant la formule 3.2, l'approximation asymptotique suivante :

$$\begin{aligned} l(n) &\sim C_1C_2\dots C_r \sum_{k_1+\dots+k_r=n} \binom{n}{k_1, k_2, \dots, k_r} \omega_1^{k_1}\dots\omega_r^{k_r} \\ &\sim C_1C_2\dots C_r(\omega_1+\dots+\omega_r)^n \end{aligned}$$

On se rappelle que notre but était de calculer la probabilité $\mathbf{Pr}(n_1,\dots,n_r)$ de la proposition 3.2.1. Avec notre approximation de $l(n)$, on a désormais asymptotiquement :

$$\mathbf{Pr}(n_1,\dots,n_r) \sim \frac{\binom{n}{n_1,\dots,n_r}\omega_1^{n_1}\omega_2^{n_2}\dots\omega_r^{n_r}}{(\omega_1+\omega_2+\dots+\omega_r)^n}$$

Qui plus est, il existe des méthodes simples pour choisir les $n_1,\dots,n_r$ avec cette probabilité. Nous avons donc une bonne approximation de $\mathbf{Pr}(n_1,\dots,n_r)$, puisque nous avons remarqué que la convergence de l'approximation des $l_i(n)$ était exponentielle. De plus, pour les petites valeurs de n, le coût du calcul exact à partir des formules 3.1 et 3.2 est acceptable.

Génération uniforme en présence de synchronisations

La présence de synchronisations dans nos composants complexifie grandement la tâche de la génération. Il faut en ce cas découper chaque composant en plusieurs parties, selon qu'elles fassent intervenir les transitions avec synchronisations ou non, et reprendre le travail précédent pour chacune. Nous essaierons tant que possible d'éviter d'avoir besoin de ces techniques en usant d'affinages heuristiques et d'idées intuitives. Le travail sur les synchronisations fut entamé dans [DGG+06] et est poursuit par les études de OUDINET *et al.* [**?**].

Marche aléatoire guidée uniforme

Une question que peut se poser le lecteur est de savoir si l'on peut parler, avec ce que l'on a vu, de *marche aléatoire guidée uniforme* dans le système, alors que nous utilisions jusqu'à présent les termes génération ou tirage uniforme.

Il faut bien voir que les marches aléatoires guidées entreprises sur les spécifications ne sont pas exécutées parallèlement, comme nous le faisions en section 2.4. Chaque marche est faite sur un composant sans se préoccuper de l'état des autres, et c'est le mélange final qui permet d'obtenir, à partir de chemins uniformément générés dans les composants, un chemin dans le système.

Lors d'une marche aléatoire sur le système entier à r composants, la probabilité de passer d'un état $(s_1,\ldots,s_r)$ à un état $(q_1,\ldots,q_r)$ est $\prod_{i=1}^{r}\mathbf{Pr}(s_i \to q_i)$. Pour obtenir une marche aléatoire uniforme sur le système, il faut donc biaiser ces probabilités pour y inclure les probabilités extraites par l'algorithme de mélange, ou plus précisément, ajouter une surcouche algorithmique pour choisir quel automate avance. Contentons-nous pour l'instant de bien voir que cela est faisable et l'on se permet donc d'évoquer le principe de marche aléatoire (guidée) uniforme dans le produit.

CHAPITRE

4 Travail effectué

Nous présentons dans cette section différentes façons de combiner l'algorithme HIT-OR-JUMP avec les résultats de la section 3.2. Nous exposons dans un premier temps quels sont nos objectifs et les difficultés que l'on pourrait entrevoir, puis nous voyons les différentes perspectives envisagées. La première d'entre elles, rapidement abandonnée, tente d'être le plus fidèle possible à l'algorithme 2 d'HIT-OR-JUMP. Les autres se rapprocheront plus volontiers de la méthode de la section 3.2 et des marches aléatoires en général.

4.1 Premières considérations

Vouloir combiner HIT-OR-JUMP et la marche uniforme définie précédemment pose un premier problème de compatibilité des algorithmes. En effet, si un algorithme a besoin d'être lancé pour obtenir un résultat, il doit aussi s'arrêter. Or nous avons d'une part HIT-OR-JUMP qui s'arrête sur un critère *qualitatif* non forcément borné en temps – nous avons parcouru assez de transitions, si ce n'est toutes – et la marche uniforme qui s'arrête sur un critère *quantitatif* borné en temps – le chemin a atteint la longueur désirée.

Notre cas d'utilisation des résultats de la section 3.2 ne nous permet pas de nous dispenser de ce dernier paramètre, nous devrons donc trouver une bonne valeur pour la taille du chemin à générer de manière à avoir une bonne espérance d'avoir couvert les transitions du module testé. Cette discussion sera faite en section 4.4 ; pour l'instant, nous admettrons que nous connaissons cette valeur que nous noterons dans le reste du document ψ. Cette longueur aura été découpée pour les r composants en $\psi = n_1 + \cdots + n_r$ par les techniques de la section 3.2.

Puisque nous devons modifier la partie aléatoire de l'algorithme HIT-OR-JUMP, nous considérerons généralement que nous n'avons pas de HIT pour, par exemple, établir quels sont nos objectifs formellement. Les HIT ne sont au final que des perturbations de notre algorithme qui peuvent intervenir à presque tout moment. Cette considération sera bien évidemment abandonnée lors de l'étude en complexité du chapitre 5.

Notre objectif est multiple. Tout d'abord, nous souhaitons, et c'est bien naturel, améliorer HIT-OR-JUMP. Nous verrons que la seule utilisation de l'uniformité ne nous permet pas for-

cément de faire cela ; pour autant, elle nous permettra d'avoir une consommation mémoire moindre, ce qui constitue un autre objectif.

Notre démarche suivra deux étapes. Dans un premier temps nous verrons comment l'intégration de la marche uniforme à HIT-OR-JUMP peut se faire, sans considérer une quelconque amélioration, puis nous verrons où il nous est possible d'agir pour obtenir un progrès supplémentaire.

4.2 Marche aléatoire sur $\mathscr{A}^\phi$

La première approche que nous exposerons, première historiquement, tend à ne modifier que la partie JUMP de l'algorithme, le parcours en profondeur n'étant remanié que par dommage collatéral. Le point de départ de cette approche est l'idée intuitive selon laquelle doivent être privilégiées les feuilles ayant les plus grands sous-arbres, en s'aidant des résultats de la section 3.2 pour le calcul de leurs tailles.

Posons tout d'abord, pour faciliter la discussion, la définition suivante :

Définition 4.2.1 (Puissance d'un ECFSM) *Soit $\mathscr{A} = \langle I, O, S, \vec{x}, T\rangle$ un ECFSM. On définit la puissance k-ième de $\mathscr{A}$, notée $\mathscr{A}^k$, par un ECFSM de mêmes ensembles d'états, de symboles et de variables que $\mathscr{A}$. Il y a une transition entre deux nœuds s et p si et seulement si :*

$$\begin{aligned} \exists (q_j)_{j\in[\![1..k]\!]} \in S^k, \quad & \forall\, j \in [\![1..k-1]\!] \langle q_j, q_{j+1}, i_j, o_j, P_j, A_j\rangle \in T \\ et \quad & \langle s, q_1, i_0, o_0, p_0, A_0\rangle \in T \\ et \quad & \langle q_k, p, i_k, o_k, p_k, A_k\rangle \in T \end{aligned}$$

Le prédicat de cette transition est alors

$$P_0(\vec{x}) \wedge P_1(A_0(\vec{x})) \wedge \ldots \wedge P_k(A_{k-1}(A_{k-2}(\ldots(A_0(\vec{x}))\ldots)),$$

l'affectation de cette transition

$$\vec{x} := A_k(A_{k-1}(\ldots(A_0(\vec{x}))\ldots)),$$

et son entrée et sa sortie sont les concaténations des i_j et o_j pour $j \in [\![0..k]\!]$.

En d'autres termes, la puissance k-ième d'un ECFSM est un automate en tout point semblable au premier sauf qu'il n'existe une transition entre deux nœuds seulement s'il existe un chemin de longueur exactement k dans l'automate de départ.

Cette définition posée, reformulons quelque peu le principe d'HIT-OR-JUMP. Dans une exécution sans HIT de l'algorithme sur l'automate produit $\mathscr{A}$, le choix aléatoire se fait sur les nœuds à une distance ϕ du nœud courant ; de fait, dans ce contexte, HIT-OR-JUMP sans HIT revient à faire une marche aléatoire sur l'automate $\mathscr{A}^\phi$.

L'idée est donc d'évaluer la probabilité de passer d'un nœud s, sommet duquel une itération d'HIT-OR-JUMP repart, à un nœud t_i, un des successeurs à distance ϕ de s, en essayant autant que possible de faire une marche uniforme sur $\mathscr{A}^\phi$.

Pour ce qui est de la pratique, rappelons-nous tout d'abord que nous choisissions lors de la marche uniforme sur un composant la probabilité d'aller d'un nœud s à un de ses successeurs t d'un composant donné par

$$\mathbf{Pr}[s \to t] = g_{m-1}(t)/g_m(s)$$

avec m la taille du chemin à générer. Nous avons évoqué dans la section 3.2 le fait que la création d'une marche uniforme dans l'automate produit faisait intervenir les probabilités du mélange uniforme en plus du produit des probabilités des transitions des composants.

La présente technique n'optimise le JUMP qu'en induisant une uniformité sur les composants $\mathscr{A}_i, i \in [\![1..r]\!]$ et non dans le produit $\mathscr{A} = \prod A_i$: la j-ième projection du chemin généré sur $\mathscr{A}$, donc un chemin sur $\mathscr{A}_j$, sera uniforme sur $\mathscr{A}_j$. De fait, nous ne considérerons pas dans un premier temps les probabilités extraites de l'algorithme de mélange.

HIT-OR-JUMP ainsi modifié suit le déroulement suivant. Lorsque l'on arrive sur un nœud s, on effectue un parcours exhaustif de profondeur ϕ. En cas de HIT, on fait la même chose que précédemment, à savoir repartir de la destination de l'arête concernée. Le changement opéré par cette version est donc sur le tirage du successeur de s dans $\mathscr{A}^\phi$, autrement dit, le choix d'une feuille de l'arbre de parcours.

Soit s le nœud courant, $t_i, i \in [\![1..k]\!]$ les feuilles de l'arbre de parcours, et π^i la fonction de projection de la i-ième composante, pour tout i. Nous serions tenté dans un premier temps d'évaluer ce qui nous intéresse par

$$\mathbf{Pr}[s \to t_i] = \prod_{j=1}^{k} \mathbf{Pr}[\pi^j(s) \to \pi^j(t_i)]. \tag{4.1}$$

Avec une pareille approche nous faisons face à deux problèmes. Le premier est lié au fait que nous choisissons un successeur pour s non pas parmi tous les nœuds à distance ϕ, mais seulement un sous-ensemble de ces nœuds. Pour illustrer ce fait, soit l'exemple suivant :

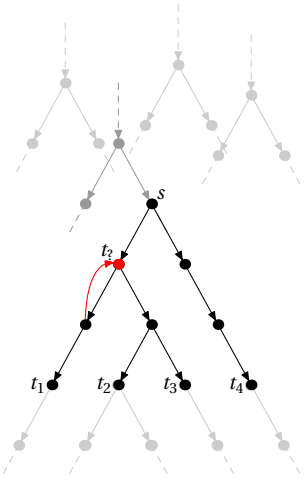

FIG. 4.1: Arc retour sur le parcours exhaustif

L'arc en arrière rouge du parcours en profondeur (avec $\phi = 3$), qui sera tout bonnement ignoré avec HIT-OR-JUMP, permet d'atteindre le nœud rouge en 3 transitions. L'ensemble $T = t_i, i \in [\![1..4]\!]$ n'est donc pas l'ensemble des successeurs de s dans $\mathscr{A}^\phi$, puisqu'il manque $t_?$. De fait, la valeur de $\mathbf{Pr}[s \to t_i]$ de l'équation 4.1 n'est pas exacte mais est une *proportion* par rapport à la somme de ces calculs pour tous les successeurs considérés.

Le deuxième problème de l'équation 4.1 est qu'elle nous oblige à sortir de notre chapeau une probabilité de *stagner*, c'est-à-dire de rester au même nœud sur un des automates composants.

En effet, mettons que $s = (s_1, \ldots, s_k, \ldots, s_r)$ et qu'un de ses successeurs soit $t = (t_1, \ldots, s_k, \ldots, t_r)$. L'équation 4.1 nous impose de connaître $\mathbf{Pr}[s_k \to s_k]$, probabilité qui peut ne pas apparaître dans le composant k s'il n'y a pas de boucles sur s_k. Ce problème surgit du fait que nous ayons pris un produit *asynchrone* d'automates, et ne peut être résolu qu'en fixant une probabilité arbitraire de rester sur un même nœud. Nuançons tout de même : ce problème serait bien moins présent si nous considérions les résultats de marche uniforme *avec* synchronisations.

Continuons sur les défauts de cette méthode. Supposons que nous nous trouvions sur un nœud s et qu'il nous reste à générer un chemin de longueur ϕ' découpé en $\phi' = n'_1 + \cdots + n'_r$. Le principal problème se trouve alors si l'on fait un HIT qui se trouve être atteint par un chemin dans le produit faisant intervenir $N > n'_k$ transitions dans le composant k. En effet, nous nous retrouverions à vouloir décrémenter n'_k de plus de sa valeur. On peut faire, bien évidemment, une décision arbitraire selon laquelle $n'_k = 0$, mais la seule solution valable pour ce problème est de supposer que le ϕ de départ est assez grand pour éviter ces situations.

Ces problèmes entachant la toute beauté des deux méthodes à fusionner, cette combinaison-ci a été rapidement abandonnée au profit d'un éloignement de l'algorithme original HIT-OR-JUMP.

4.3 Marche aléatoire sur $\mathscr{A}$ avec plusieurs tentatives sur non-HIT

De l'équivalence d'un parcours en profondeur et des marches aléatoires

Nous avions, dans HIT-OR-JUMP, une partie entièrement déterministe : le parcours exhaustif (que l'on supposera en profondeur dans le reste de l'étude). Nous avons vu dans la section précédente combien cette touche de déterminisme pouvait coûter au bon fondement de la combinaison d'HIT-OR-JUMP avec la marche uniforme.

Une des idées principales de cette section est de se passer du parcours exhaustif, tout en tendant à l'émuler afin de garder les avantages qu'elle apportait. Afin de s'inscrire plus aisément dans le principe de marche uniforme, le parcours sera remplacé par plusieurs marches (dont la nature sera à définir).

La méthode est la suivante. Nous effectuions précédemment un parcours en profondeur de taille bornée ϕ à partir d'un nœud s, dont l'arbre associé avait M feuilles, chacune d'entre elles correspondant à un chemin dans l'arbre. En lieu et place de ce parcours, $k.M$ marches de tailles bornées par ϕ sont effectuées à partir s. Le facteur k dépend des degrés sortants des nœuds dans l'arbre, mais il devient constant si les marches effectuées dans cet arbre sont uniformes.

Nous utiliserons cette correspondance intuitive dans le reste de cette section, et le nombre de marches à effectuer à partir du nœud s sera noté N jusqu'à ce qu'en section 4.3 on s'intéresse à étudier sa valeur. Intuitivement, ce nombre doit nous apporter une certaine confiance dans le fait que nous fassions un travail similaire au parcours exhaustif.

Reprise sur non-HIT

Deux questions restent en suspens, celle du comportement à adopter lors d'un HIT, et que faire lors d'un JUMP. Sans considérer pour l'instant la marche uniforme, mais en continuant sur notre voie de HIT-OR-JUMP probabiliste, l'algorithme suivant répond à ces questions :

Algorithme 4 HIT-OR-JUMP-RW

ENTRÉES: Les composants $\mathscr{A}_i, i \in [\![1..r]\!]$ dont $\mathscr{A}_k$ à tester, $\phi \in \mathbb{N}^*$, et N précédemment défini
SORTIES: Une séquence de test

```
s ← vecteur des états initiaux des automates
σ ← ""
tant que toutes les transitions de A_k ne sont pas visitées faire
  Π ← ∅ {Ensemble des chemins tirés}
  pour i := 1 à N faire
    Tirer un chemin π de longueur φ à partir de s dans le produit,
    Π ← Π ∪ {π}
    si π comporte une transition τ contenant une transition non marquée de A_k alors
      {Partie HIT }
      σ ← σ + entrées des transitions de s à τ
      Marquer la transition empruntée de A_k
      s ← destination de la transition τ
    fin si
  fin pour
  {Partie JUMP }
  Choisir aléatoirement de manière uniforme un chemin π dans Π
  σ ← σ + entrées des transitions de π
  s ← dernier nœud de π
fin tant que
Retourner σ
```

Comme l'algorithme le montre bien, l'avantage de cette version n'est pas clair. Le fait, principalement, que l'on soit obligé de se rappeler de toutes les marches effectuées semble ramener cet algorithme au HIT-OR-JUMP classique, mais rappelons-nous que ceci était bien notre but. Pour enfoncer le clou, il est bel et bien obligatoire de se rappeler de toutes les marches effectuées pour permettre de faire un choix similaire au JUMP de l'algorithme original ; en effet, si l'on prenait, par exemple, le dernier chemin tiré, nous n'aurions pas un saut uniforme dans $\mathscr{A}^\phi$, donc pas d'échappatoire au problème du goulot d'étranglement.

Nous verrons en section 4.5 comment se défend cette version en pratique.

Ajout de la marche uniforme

Nous avons présenté en section précédente une version probabiliste d'HIT-OR-JUMP, à ceci près que nous faisions des reprises sur non-HIT. Ajouter les résultats de marche uniforme dans ce cadre semble facile. On rappelle pour cela l'analogue de l'équation 4.1, dans le cadre de la marche non plus sur $\mathscr{A}^\phi$ mais sur $\mathscr{A}$. Pour un nœud s du produit de successeurs directs t_i,

$$\mathbf{Pr}(s \to t_i) = \prod_{j=1}^{k} \mathbf{Pr}(\pi^j(s) \to \pi^j(t_i)). \tag{4.2}$$

Les probabilités de droite étant données par les résultats de la section 3.2 comme étant la proportion de chemins de taille $n-1$ issues de t_i sur le nombre de chemins de taille n issues de s. L'algorithme s'énonce alors comme suit :

Algorithme 5 HIT-OR-JUMP-UW

ENTRÉES: Les composants $\mathscr{A}_i, i \in [\![1..r]\!]$ dont $\mathscr{A}_k$ à tester, $\phi \in \mathbb{N}^*$, et N et ψ précédemment définis
SORTIES: Une séquence de test
  $s \leftarrow$ vecteur des états initiaux des automates
  $\sigma \leftarrow$ ""
  Découper $\psi = n_1 + \cdots + n_r$
  Calculer les probabilités des transitions dans les composants,
  tant que toutes les transitions de $\mathscr{A}_k$ ne sont pas visitées et $\psi > 0$ **faire**
    pour $i := 1$ **à** N **faire**
      Tirer un chemin π de longueur ϕ à partir de s dans le produit avec les probabilités évoquées,
      si π comporte une transition τ contenant une transition non marquée de $\mathscr{A}_k$ **alors**
        {Partie HIT }
        $\sigma \leftarrow \sigma +$ entrées des transitions de s à τ
        Marquer la transition empruntée de $\mathscr{A}_k$
        Mettre à jour les n_i et ψ en fonction du nombres de transitions parcourues sur les composants,
        $s \leftarrow$ destination de la transition τ
      fin si
    fin pour
    {Partie JUMP, π est le dernier chemin tiré}
    $\sigma \leftarrow \sigma +$ entrées des transitions de π
    Mettre à jour les n_i et ψ en fonction du nombres de transitions parcourues sur les composants,
    $s \leftarrow$ dernier nœud de π
  fin tant que
  Retourner σ

On observe tout d'abord la suppression de l'ensemble Π ; il s'agit plus d'un principe intuitif qu'autre chose. En effet, il serait tout à fait légitime de le supprimer si nous tirions un chemin uniforme de taille ϕ à chaque itération d'HIT-OR-JUMP. Or ce qui est ici fait, est d'interrompre une exécution censée nous conduire à un message uniforme de taille ψ sur le produit. Nous verrons en pratique, dans la section 4.5, si cette opposition entre uniformité locale et globale change les performances de l'algorithme.

Influence vers les transitions non parcourues

Rappelons que les transitions du produit sont empruntées avec une probabilité qui est le produit des probabilités dans les composants. Un de ces composants nous intéresse tout particulièrement, $\mathscr{A}_k$, le composant sous test.

Les valeurs de la formule 3.1 permettent de trouver les probabilités amenant à une marche uniforme sur le composant. Nous proposons dans cette section de recalculer à chaque itération d'HIT-OR-JUMP-UW ces valeurs pour le composant $\mathscr{A}_k$ de manière à influencer la marche vers des transitions « intéressantes ».

Plus précisément, posons, au début d'une itération d'HIT-OR-JUMP-UW, T_{np} l'ensemble des transitions non parcourues dans $\mathscr{A}_k$. Au lieu de compter, comme le fait la formule 3.1, le nombre de chemins issus d'un nœud, on prendra pour valeur le nombre de chemins issus d'un nœud *qui contiennent une transition de* T_{np}. Voici le calcul :

$$\begin{aligned} g_0(s) &= 0 \\ g_i(s) &= \sum_{s' \text{ successeur de } s} \begin{cases} 1 + g_{i-1}(s') & \text{si } s \to s' \in T_{np} \\ g_{i-1}(s') & \text{sinon.} \end{cases} \quad \text{pour } i \in [\![1..n_k]\!] \end{aligned} \tag{4.3}$$

L'ensemble T_{np} évoluant à chaque HIT, le calcul de cette fonction est à refaire un nombre de fois en $\mathscr{O}(|T_{A_k}|)$, avec T_{A_k} les transitions de A_k. Encore une fois, nous verrons en section 4.5 ce qui attend cette version, nommée HIT-OR-JUMP-UW', d'un point de vue performance.

Calcul de N

Trouver une valeur exacte de N à chaque itération est une tâche d'autant plus fastidieuse qu'elle n'est pas réalisable. En effet, quelques soient nos calculs, il faut bien voir que nous effectuons N marches *aléatoires*, ce qui signifie bien qu'un nombre donné de marche ne nous apporte qu'une certaine *confiance* en le résultat.

Une méthode désormais classique pour répondre à ce genre de contrainte est issue du *model-checking*. Il s'agit d'une partie de la méthode dite de MONTE-CARLO [GS05, Section 3] qui s'applique comme suit.

Soit p la probabilité d'atteindre une transition non marquée de A_k en ϕ coups. Autrement dit, p est la probabilité qu'une itération de l'algorithme fasse un HIT. Après M marches bornées de profondeur ϕ, la probabilité d'atteindre une transition non marquée est de :

$$F(M) = \sum_{i=1}^{M} (1-p)^{i-1}.p = 1 - (1-p)^M.$$

Avec X la variable aléatoire géométrique dont la valeur est le nombre d'essais jusqu'à ce qu'un HIT possible soit fait, il est clair que $F(M) = \mathbf{Pr}[X \leq M]$. Posons δ notre paramètre de confiance, on cherche alors à avoir $F(M) \geq 1 - \delta$. On obtient donc que

$$M \geq \frac{\log(\delta)}{\log(1-p)}.$$

Cela étant, dans notre cas p est inconnu. De fait, en plus de supposer que $F(M) \geq 1-\delta$, on suppose que $p \geq \epsilon$, ce qui amène que

$$N \geq \frac{\log(\delta)}{\log(1-\epsilon)} \geq \frac{\log(\delta)}{\log(1-p)}.$$

Ceci nous donne donc avec un paramètre de confiance δ une borne inférieure sur le nombre d'essais N nécessaires pour atteindre un HIT possible, donc :

$$\mathbf{Pr}[\text{Erreur}] = \mathbf{Pr}[X > N | p \geq \epsilon] < \delta.$$

4.4 Calcul de ψ

L'état actuel du calcul de cette composante est simple. Nous avons extrait des calculs dont la viabilité ne saute pas aux yeux, pas plus que leur exactitude. Ces calculs n'étant pour l'instant pas achevé, nous choisissons pour l'instant, très arbitrairement, de récupérer comme valeur de ψ une quantité proportionnelle à la taille du chemin retournée par l'algorithme HIT-OR-JUMP originel.

Nous espérons avancer rapidement sur ce calcul, pour permettre de trouver des résultats théoriques plus probants.

4.5 Expériences

Les expériences empiriques actuelles se sont principalement focalisés sur la vérification du fait que les algorithmes implémentés le soient correctement. Ont été implémenté :

- Le HIT-OR-JUMP classique (algorithme 2),
- Le HIT-OR-JUMP probabiliste sans marche uniforme (algorithme 4),
- Le HIT-OR-JUMP probabiliste avec marche uniforme (algorithme 5).

Le cas d'utilisation de ces implémentations a été le protocole du bit alterné, dans lequel trois modules sont présents :

1. L'émetteur, qui envoie un bit à 0 puis un bit à 1,
2. Le récepteur, qui reçoit ce bit et envoie un acquittement à 0 ou 1 selon le cas,
3. Le moyen de transport, qui fait passer un message de l'émetteur au récepteur en introduisant possiblement des erreurs.

L'émetteur doit donc envoyer un bit à 0 ou 1 tant qu'il n'a pas reçu le bon acquittement, tandis que le récepteur informe d'une erreur tant qu'il n'a pas reçu ce même bit.

Les expériences conduites sur cet exemple-jouet ne permettent pas pour l'instant de juger de l'efficacité des algorithmes, mais c'est un futur proche qui nous offrira l'opportunité de comparer les performances.

CHAPITRE

5 Ouverture

Ce chapitre, loin d'être terminé, s'axera sur deux sujets distincts. Dans un premier temps, nous faisons une ouverture sur les possibilités d'étude en complexité de l'algorithme HIT-OR-JUMP originel. Nous voyons ensuite comment les résultats du chapitre précédent peuvent être appliqués à des notions plus générales que les ECFSM.

5.1 Étude en complexité

Une étude menée conjointement avec l'équipe d'algorithmique du LRI[1] a permis d'établir des premiers liens entre la complexité d'HIT-OR-JUMP et des éléments connus de complexité dans le domaine des marches aléatoires.

Toutefois, les résultats courants ne sont pas encourageants ; nous n'avons en effet pas encore pu estimer une borne maximale pour l'algorithme lui-même, mais seulement pour des approximations de celui-ci.

L'avancement, lent dans ce domaine, n'est pas voué à aboutir, du moins dans le laps de temps qui nous est accordé. Nous gardons cependant bon espoir de la possibilité d'un déblocage, le problème ayant été posé à des pointures de l'équipe Algo – impliquant, hélas, des délais de réponses bien plus importants.

Une version plus avancée de ce rapport, dans la première moitié de 2008, nous permettra peut-être d'en apprendre plus sur ces résultats.

5.2 Généralisation

Il est sûrement très pertinent d'utiliser la démarche de ce document, et ses résultats, dans d'autres domaines que celui de la vérification ; cette navigation pseudo-uniforme n'ayant pas encore révélé tous ses secrets, il y aura bien plus à dire sur cette possibilité dans les mois à venir.

[1] Équipe Algo LRI : `http ://algo.lri.fr`

Conclusion

Le test de conformité de composants imbriqués est un secteur sur lequel se portent de plus en plus les intérêts des entreprises. Il s'agit en effet de s'assurer du bon fonctionnement de processus, souvent critiques. Il est donc nécessaire de créer et d'améliorer les techniques pour offrir un service toujours plus efficace.

Dans un domaine en pleine évolution, où les innovations sont quotidiennes, nous avons cru bon apporter une nouvelle pierre à l'ouvrage. En mélangeant des techniques de deux domaines distincts, les performances d'un algorithme ont pu être intuitivement améliorées. Il s'agissait ainsi d'offrir une alternative encore plus viable au test de conformité exhaustif.

Lors du tests de composants, la principale difficulté est et reste le manque de connaissance sur le système global. Il est crucial d'éviter de construire le système global complet, mais il nous est toutefois nécessaire d'en connaître des propriétés précises pour faciliter son étude. Le présent travail assure un critère de couverture très pertinent pour le tirage de séquences de test dans ces modèles.

En assurant une couverture des exécutions possibles du système qui soit proche de l'uniformité, ce travail permet l'extraction de séquences de test ayant plus de chance d'atteindre des cas d'utilisation extrêmes ou rares, vérifiant ainsi des portions de code peu employées.

Cette avancée, qui n'est pas à dénigrer, n'est pour autant qu'une première approche dans le travail à l'intersection de la vérification et de l'exploration de système. Il s'agit désormais d'approfondir ces travaux, et d'essayer la même approche, avec peut-être plus de succès, avec d'autres résultats de ces domaines.

Bibliographie

[Bri88] Ed BRINKSMA. « A theory for the derivation of tests ». *Proc. IFIP WG6.1 8th Int. Symp. on Protocol Specification, Testing and Verification*, 1988.

[CGP00] E.M. CLARKE, O. GRUMBERG et D. PELED. *Model Checking*. MIT Press, 2000.

[CLRZ99] Ana R. CAVALLI, David LEE, Christian RINDERKNECHT et Fatiha ZAÏDI. « Hit-or-Jump : An algorithm for embedded testing with applications to IN services ». Dans *FORTE XII / PSTV XIX '99 : Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 41–56, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[DGG+06] Alain DENISE, Marie-Claude GAUDEL, Sandrine-Dominique GOURAUD, Richard LASSAIGNE et Sylvain PEYRONNET. « Uniform random sampling of traces in very large models ». Dans *RT '06 : Proceedings of the 1st international workshop on Random testing*, pages 10–19, New York, NY, USA, 2006. ACM Press.

[FS06] Philippe FLAJOLET et Robert SEDGEWICK. *Analytic Combinatorics*. Cambridge University Press, April 2006. Chapters I–IX of a book to be published by Cambridge University Press, 717p.+x, available electronically from P. Flajolet's home page.

[FZC94] Philippe FLAJOLET, Paul ZIMMERMAN et Bernard Van CUTSEM. « A calculus for the random generation of labelled combinatorial structures ». *Theor. Comput. Sci.*, 132(1-2) :1–35, 1994.

[GM03] Benoît GAUDIN et Hervé MARCHAND. « Contrôle de systèmes à événements discrets hiérarchiques ». Dans *4ième Colloque Francophone sur la Modélisation des Systèmes Réactifs, MSR'03*, Metz, France (Version Francaise de ECC'03), October 2003.

[Gol97] Oded GOLDREICH. « Combinatorial Property Testing (a survey). ». *Electronic Colloquium on Computational Complexity (ECCC)*, 4(56), 1997.

[GS05] Radu GROSU et Scott A. SMOLKA. « Monte Carlo Model Checking ». Dans *TACAS*, pages 271–286, 2005.

[HC83] Timothy HICKEY et Jacques COHEN. « Uniform Random Generation of Strings in a Context-Free Language ». *SIAM Journal on Computing*, 12(4) :645–655, 1983.

[JVV86] Mark JERRUM, Leslie G. VALIANT et Vijay V. VAZIRANI. « Random Generation of Combinatorial Structures from a Uniform Distribution. ». *Theor. Comput. Sci.*, 43 :169–188, 1986.

[LS82] Simon S. LAM et A. U SHANKAR. « Protocol Verification Via Projections ». Rapport Technique, Austin, TX, USA, 1982.

[LSKP96] D. LEE, K.K. SABNANI, D.M. KRISTOL et S. PAUL. « Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach ». 44(5) :631–640, 1996.

[LY96] David LEE et Mihalis YANNAKAKIS. « Principles and Methods of Testing Finite State Machines - A Survey ». Dans *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[Mil95] Robin MILNER. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[MR96] Rajeev MOTWANI et Prabhakar RAGHAVAN. « Randomized algorithms ». *ACM Comput. Surv.*, 28(1) :33–37, 1996.

[Pet62] Carl Adam PETRI. *Kommunikation mit Automaten*. Bonn : Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Sei72] Charles SEITZ. « An approach to design checking experiments based on a dynamic model ». *Theory of machines and computations*, 1972.

[Sin93] Alistair SINCLAIR. *Algorithms for random generation and counting : a Markov chain approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.

[Yan91] Mihalis YANNAKAKIS. « Testing finite state machines ». Dans David LEE, éditeur, *STOC '91 : Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 476–485, New York, NY, USA, 1991. ACM Press.

Table des figures

Table des algorithmes

Index

Les numéros de pages en gras indiquent la définition du terme ou la référence principale.