Hit-or-Jump : Un algorithme pour le test imbriqué avec des applications aux services R.I.

Ana Cavalli* , **David Lee†** , **Christian Rinderknecht*** **et Fatiha Zaïdi**[1]

** Institut National des Telecommunications*
9 rue Charles Fourier
F-91011 Evry Cedex
{Ana.Cavalli, Christian.Rinderknecht, Fatiha.Zaidi}@int-evry.fr
† Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974, USA lee@research.bell-labs.com

RÉSUMÉ — Cet article présente un nouvel algorithme, Hit-or-Jump, pour le test imbriqué de composants de systèmes de communication qui peuvent être modélisés par des machines à états finis étendues communicantes. Il construit efficacement des séquences de test avec une couverture de fautes élevée. Il ne donne pas lieu à l'explosion du nombre d'états, telle qu'elle se produit dans la recherche exhaustive, et il couvre rapidement les composants du système sous test sans être "attrapé", tel que cela se produit pour les recherches aléatoires. En outre, il est une généralisation et une unification à la fois de la recherche exhaustive et des recherches aléatoires. Les deux sont des cas particuliers de Hit-or-Jump. L'algorithme a été implanté et appliqué au test imbriqué des services téléphoniques dans une architecture de réseaux intelligents, incluant le Basic Call Service (BCS), et cinq services supplémentaires: Originating Call Screening (OCS), Terminal Call Screening (TCS), Call Forward Unconditionnal (CFU), Call Forward on Busy Line (CBL) et Automatic Call Back (ACB).
MOT-CLÉS : *test de conformité, test imbriqué, machines à états finis étendues communicantes*

1. Introduction

Du fait de l'essor de l'informatique et d'une demande croissante des utilisateurs pour obtenir des services sophistiqués, les systèmes de protocole de communication sont devenus de plus en plus complexe et de fait encore moins fiable. Le test de conformité, qui garantit que les implantations d'un protocole sont correctes, est devenu indispensable pour le développement de systèmes de communications fiables. Les méthodes de test traditionnelles consistent à tester ces systèmes entièrement soit à tester ses composants en isolement. Or le test exhaustif de ces systèmes devient difficile du fait de leur taille importante. D'autre part, le test en isolement des composants n'est pas toujours possible, en partie à cause des interactions possibles entre les composants au sein du système. Le test imbriqué, ou test en contexte, est devenu ces dernières années un axe de recherche important dans le domaine du test de conformité. L'objectif du test imbriqué est de tester si une

implantation du composant du système est conforme à sa spécification dans le contexte des autres composants. On suppose généralement que le testeur n'a pas de lien direct avec le composant sous test; l'accès à ce composant se fait en passant par les autres composants du système. Selon le standard: «si le contrôle et l'observation sont appliqués à une ou plusieurs implantations qui sont au dessus du protocole à tester, les méthodes de test sont dites imbriquées» [9].

Différentes approches pour le test imbriqué ont été proposées dans la littérature. Elles sont basées sur les modèles de fautes [15], sur la réduction du problème au test du composant en isolement [16], sur la minimisation de la suite de test [11, 13, 18, 1], sur la couverture de faute [19], sur le test de systèmes avec des interfaces semi-controllables et non-controllables [5], ou sur le test à la volée [6]. La plupart de ces approches reposent sur les graphes d'accessibilité pour modéliser les comportements conjoints de tous les composants du système, et sont confrontées au problème bien connu de l'explosion du nombre d'états.

Notre objectif est de tester des parties spécifiées d'un composant système qui est imbriqué dans un système de communication complexe. Les parties spécifiées sont déterminées par des besoins pratiques ou par des nécessités de certification du système. Par exemple, pour un composant système donné, on peut vouloir tester toutes les transitions ou certaines valeurs limites des variables du système. On peut d'abord construire le graphe d'accessibilité, qui est le produit cartésien de tous les composants systèmes impliqués, et ensuite dériver un test qui couvre toutes les parties spécifiées du composant sous test. Malheureusement il est souvent impossible de construire le graphe d'accessibilité exhaustif pour les systèmes, étant donné l'explosion du nombre d'états. Pour éviter ce problème les recherches aléatoires ont été proposées; à tout moment, on garde seulement trace des états courants de tous les composants et on détermine le prochain état du test aléatoirement. Cette approche évite l'explosion du nombre d'états, mais elle peut tester plusieurs fois des parties couvertes et prendre beaucoup de temps pour se déplacer vers des parties non testées.

On propose une nouvelle technique: Hit-or-Jump. Elle est une généralisation et une unification à la fois de la recherche exhaustive et des recherches aléatoires, sans avoir pour autant les inconvénients des deux approches. L'essence de notre approche est la suivante. À tout moment on conduit une recherche locale à partir de l'état courant dans le voisinage du graphe d'accessibilité. Si une partie non testée est trouvée (un touché, ou *Hit*), on test cette partie et on continue le traitement à partir de là. Sinon on se déplace de manière aléatoire au frontière du voisinage recherché (un saut, ou *Jump*), et on continue le traitement à partir de là. Cette procédure évite la construction du graphe d'accessibilité. De ce fait l'espace requis est déterminé par l'utilisateur (la recherche locale), et il est indépendant des systèmes considérés. D'autre part, une recherche aléatoire peut être «attrapée» dans une certaine partie du composant sous test [11]. Notre algorithme est conçu pour «sauter» hors de la trappe et ainsi poursuivre plus en avant son exploration.

L'algorithme Hit-or-Jump a été appliqué au test imbriqué de services du réseau téléphonique. Avec l'aide de l'outil ObjectGEODE, cette étude de cas porte sur un système réel, qui a été spécifié en utilisant le langage SDL. Il décrit des services téléphoniques sur une architecture de réseau intelligent (IN). Avec le service de base Basic Call Services (BCSs), cinq autres services sont présents:Originating Call Screening (OCS), Terminating Call Screening (TCS), Call Forward Unconditional (CFU), Call Forward on Busy Line (CBL) and Automatic Call Back (ACB).

L'article est organisé de la manière suivante. La section 2 introduit les concepts de base et la testabilité des composants imbriqués. La section 3 décrit l'algorithme de génération de test Hit-or-Jump pour les composants imbriqués. La section 4 détaille les implantations. La section 5 relate les résultats expérimentaux obtenus sur une architecture IN. Enfin la section 6 conclut l'article par des remarques sur la généralisation et les variations de l'algorithme, ainsi que par des perspectives.

2. Préliminaires

Dans ce travail nous utilisons les machines à états finis pour modéliser les composants du système: l'environnement, les composants sous test et leurs implantations. Notre technique peut

également s'appliquer à d'autres modèles mathématiques, tels que les systèmes de transition [14], les réseaux de Petri [17], et les systèmes de transitions étiquetées [2].

Définition 1. Une machine à états finis étendue (EFSM) est un quintuplet $M = (I, O, S, \vec{x}, T)$ où I, O, S, $\vec{x}$, et T sont respectivement des ensembles finis de symboles d'entrées, de symboles de sorties, d'états, de variables et de transitions. Chaque transition t dans l'ensemble T est un sextuplet :

$$t = (s_t, q_t, a_t, o_t, P_t, A_t)$$

où s_t, q_t, a_t, et o_t sont respectivement l'état de départ (courant), l'état final (prochain), une entrée et une sortie. $P_t(\vec{x})$ est un prédicat sur les valeurs courantes des variables et $A_t(\vec{x})$ définit une action sur les valeurs des variables.

Initialement la machine se trouve dans un état $s^{(0)} \in S$ avec les valeurs des variables $\vec{x}^{(0)}$. Supposons que la machine se trouve dans l'état s_t avec les valeurs de variables courantes $\vec{x}$. Sur l'entrée a_t, si $\vec{x}$ est valide pour P_t, i.e. $P_t(\vec{x}) = \mathsf{true}$, alors la machine suit la transition t, émet o_t, change les valeurs courantes des variables par l'action $\vec{x} := A_t(\vec{x})$, et se place dans l'état q_t.

On modélise l'environnement et le composant sous test par des EFSM. Pendant l'exécution les états et les valeurs des variables peuvent être déterminés comme ils le sont dans la construction des graphes d'accessibilité des EFSM [8, 12]. À partir de maintenant on utilise la notation suivante : C est l'EFSM contexte, A est l'EFSM de la spécification sous test, et B est l'implantation de A. Les machines A et B communiquent de façon synchrone avec la machine C. On représente A dans le contexte de C par la notation suivante : $C \times A$. On veut tester la conformité de B par rapport à A dans le contexte de C, où C et A sont connues et B est une boîte noire. On remarquera que $C \times A$ peut ne pas être minimisée ou fortement connexe même si C et A le sont. Elles peuvent être aussi partiellement spécifiées (incomplètes).

En général, il est pas toujours possible de tester l'isomorphisme des composants imbriqués, même dans le cas des FSM. Supposons que A et B sont des FSM. On exprime l'isomorphisme de la machine par $A \cong B$. Alors nous avons:

Proposition 1. $B \cong A$ implique $C \times B \cong C \times A$. Cependant l'inverse n'est pas vrai en général.

La première partie de la proposition est triviale. Nous montrons la seconde par un contre-exemple:

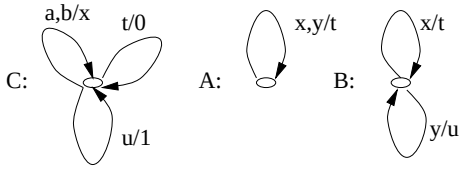

FIG. 1 -

Dans la figure 1 a et b sont des signaux d'entrée externes, 0 et 1 des signaux de sortie externes, et x, y, t, u sont des signaux d'entrée/sortie internes. Trivialement $C \times B \cong C \times A$, mais pas $B \ncong A$. Par conséquent, il est impossible de tester $A \cong B$ dans le contexte de C.

En pratique, ce que l'on veut c'est que B se comporte correctement dans le contexte de C. Ce qui revient à $C \times B \cong C \times A$. Par conséquent, le problème se réduit à tester si $C \times B \cong C \times A$. Cependant le vrai objectif est de tester le composant A, en supposant que la machine de contexte C est correctement implantée. Supposons que l'on teste A en isolement. Alors on peut vouloir tester toutes les transitions de A. Ainsi on veut obtenir une séquence de test telle que toutes les transitions de A soient exécutées. De manière similaire, dans le test imbriqué, on veut obtenir une séquence de test (avec les entrées externes) telle que toutes les transitions du composant A soient exécutées. On veut précisément dériver des tests pour $C \times A$ tels que toutes les transitions de A soient testées. On peut vouloir différente couverture de A, plutôt que de tester toutes les transitions. Par exemple, on veut souvent tester les bornes des valeurs de variables. En général, on veut obtenir une séquence de test pour $C \times A$, i.e. pour tester le composant A dans le contexte

de C, telle que la machine composant A soit couverte selon le critère spécifié. Ce critère peut être spécifié par l'affectation d'une couleur distincte à chaque entité (transition ou valeur d'une variable, par exemple) à tester et par couvrir toutes les couleurs affectées par la génération de séquences de test. D'autre part on ne se préoccupe pas de la couverture de C, puisqu'il est censé être correctement implanté. Par souci de simplification, nous supposons que le système sous test n'a pas de blocages ni de circuits vivaces, qui sont des sujets bien connus dans le domaine de la validation [8].

3. Test imbriqué

L'algorithme Hit-or-Jump est une généralisation et une unification de trois méthodes, un algorithme structuré, une recherche aléatoire, une recherche aléatoire guidée, ces algorithmes ont été présentés dans [11]. Il s'inscrit donc dans cette classe d'algorithme. Pour simplifier, on décrit la procédure qui permet de couvrir toutes les transitions de la machine composant sous test. En pratique c'est un critère qui est communément employé. Comme nous l'avons précisé précédemment, d'autres critères de couverture peuvent être réduit à la couverture de couleur du composant sous test, et notre procédure peut être facilement adaptée pour la génération de tests pour les couleurs couvertes, il s'agit tout simplement d'un processus de marquage.

3.1. L'algorithme Hit-or-Jump

Les problèmes avec les recherches aléatoires présentées dans [11] sont : (1) Etre attrapé dans un petit voisinage; (2) Avoir une faible probabilité de traverser un «pont étroit»pour tester les parties au-delà du pont; et (3) Oublier les transitions non marquées de A, même si elles sont toutes proches (plus d'un pas à partir du nœud courant). L'algorithme Hit-or-Jump a été conçu pour éviter ces écueils, sans construire le graphe d'accessibilité. Il ne nécessite pas non plus la construction du graphe d'accessibilité de $C \times A$, et l'exécution est plus efficace que les recherches aléatoires pures. La recherche locale est utilisée dans la procédure, et peut être une recherche en profondeur d'abord ou largeur d'abord.

L'ALGORITHME HIT-OR-JUMP

conditions initiales. La machine contexte C se trouve dans l'état initial $s_C^{(0)}$, la machine composant sous test A se trouve dans l'état initial $s_A^{(0)}$ et les variables du système ont les valeurs initiales $\vec{x}^{(0)}$.

terminaison. L'algorithme termine quand toutes les couleurs (transitions) de A sont marquées.

exécution.

1. À partir du nœud courant $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$ conduire une recherche dans $C \times A$ jusqu'à :
   (a) Atteindre une arête qui est associée à des couleurs non marquées de la machine composant A : un **HIT** (touché). Alors :
      i. Inclure le chemin du nœud courant à l'arête (incluse) dans la séquence en construction;
      ii. Marquer la nouvelle couleur exécutée de A;
      iii. Arriver au nœud $(s_C^{(k+1)}, s_A^{(k+1)}, \vec{x}^{(k+1)})$;
      iv. Détruire le graphe de recherche;
      v. Répéter à partir de 1.

   ou

   (b) Atteindre une profondeur de recherche ou limite d'espace sans avoir rencontré aucune couleur non marquée de A : Un **JUMP** (saut), alors :
      i. On a construit un arbre de recherche, ayant pour racine $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$.

ii. Examiner toutes les feuilles de l'arbre et en choisir un uniformément de manière aléatoire.
iii. Inclure le chemin partant de la racine jusqu'à la feuille choisie dans la séquence de test.
iv. On arrive à la feuille choisie $(s_C^{(k+1)}, s_A^{(k+1)}, \vec{x}^{(k+1)})$: (d'où le saut).
v. Recommencer à partir de 1.

Supposons que la profondeur de recherche locale soit à 1. Alors, évidemment Hit-or-Jump devient une recherche aléatoire. Si on renforce les priorités, alors il devient une recherche aléatoire guidée. Par ailleurs si on ne met pas de borne à la recherche, alors on construit le graphe d'accessibilité dans le pire des cas; Hit-or-Jump devient un algorithme structuré. Par conséquent, Hit-or-Jump est une généralisation des recherches aléatoires et des recherches aléatoires guidées et aussi de l'algorithme structuré. En outre, cette technique unifie ces trois approches en apparence complètement différentes.

4. L'implantation de Hit-or-Jump

Dans cette section, nous décrivons l'implantation de Hit-or-Jump. Il n'est pas possible d'utiliser l'outil ObjectGEODE seul, parce que notre algorithme nécessite la manipulation de caractéristiques de l'outil, qui ne sont pas très utilisées (et par conséquent peu documentées). Aussi on a donc développé un outil logiciel qui pilote le simulateur ObjectGEODE. Par souci de simplification, on ne décrit pas ici les détails, les lecteurs intéressés peuvent se reporter à [3]. Notre objectif est d'obtenir une séquence de test dans l'automate déployé, correspondant à un chemin commençant à l'état initial, qui contienne toutes les transitions du composant imbriqué sous test, sans pour autant construire l'automate déployé complet.

Interface. On fournit à notre outil les informations suivantes. Le résultat est un fichier contenant la séquence de test pour le composant imbriqué. La séquence est une série de paires d'entrées et de sorties. (1) Une disjonction de conditions d'arrêt modélise l'ensemble des transitions du composant imbriqué. Il définit le système imbriqué. (2) Un entier positif représente la profondeur limite, qui est fourni au simulateur; on s'arrête lorsque la recherche (BFS, DFS, B-DFS (DFS avec profondeur bornée)) atteint cette profondeur. (3) Les entrées que le simulateur peut exécuter à partir de l'environnement afin de simuler le système entier (ici opposé à «système imbriqué»). (4) Les variables, dépendantes du protocole, pour le simulateur (les clauses «let», comme le nombre d'actions pour les utilisateurs, le nombre maximal d'actions par utilisateur, les souscriptions aux services etc.) (5) Un scénario initial qui active tous les processus, et les met ensuite dans leurs états initiaux.

Configuration. La première étape de notre outil est de configurer et de produire 3 fichiers de démarrage, qui vont être utilisés pour piloter le simulateur. (1) main.startup. Il charge les entrées que le simulateur peut activer à partir de l'environnement, la disjonction de conditions d'arrêt les variables dépendantes du protocole, le scénario courant et spécifie le mode exhaustif de la simulation. Ensuite il lance le simulateur. (2) stop_search.startup. Ce fichier sert à retrouver la condition d'arrêt dans la disjonction qui a réellement stoppé la simulation. (3) final.startup. Il charge les entrées nécessaires au simulateur à partir de l'environnement et rejoue le scénario final, que l'on a obtenu après avoir rencontré toutes les couleurs (transitions) du composant imbriqué. Comme résultat on obtient un fichier d'historique d'ObjectGEODE à partir duquel on extrait la séquence de test.

Simulation. On commence la simulation avec le fichier main.startup; deux situations sont possibles: (1) *le simulateur fournit un scénario.* Ce fichier est produit si et seulement si on a atteint une transition non couverte du système imbriqué. Alors deux cas peuvent se produire: (A) *Il reste une seule condition d'arrêt.* Alors on a terminé l'algorithme, on lance le simulateur avec le fichier final.startup et on extrait à partir du fichier d'historique la séquence de test. (B) *Il reste au moins*

deux conditions d'arrêt. Alors on lance à nouveau le simulateur avec le fichier stop_search.startup, afin d'identifier la transition touchée parmi la disjonction courante (i.e. l'ensemble des transitions non couvertes). Ce fichier imprime l'état des conditions d'arrêt et identifie celle qui est mise à true. (2) *Le simulateur ne produit pas de scénario.* Cela signifie que le simulateur s'est arrêté sur la profondeur limite. En d'autres termes, il n'a trouvé aucune transition qui satisfasse une des conditions d'arrêt dans la disjonction. On obtient cependant un fichier, contenant l'automate déployé partiel, comme résultat de la simulation interrompue, mais on ne connaît ni l'etat courant dans la EFSM (spécification SDL), ni le chemin à partir de l'état intial. Ainsi on analyse syntaxiquement l'automate déployé et on conduit une DFS. On choisit de façon aléatoire et uniforme une feuille et on trouve le chemin (le plus court) de l'état courant à la feuille. On ajoute le chemin ainsi obtenu à la fin du scénario construit et on reprend la simulation.

5. Étude de cas : Les services téléphoniques du réseau intelligent

Dans cette section, on relate les résultats expérimentaux obtenus de lors l'application de la technique de génération de test Hit-or-Jump sur des services téléphoniques de réseau intelligent. Le service intègre des services supplémentaires: la liste noire au départ (OCS), la liste noire à l'arrivée (TCS), le transfert d'appel inconditionnel (CFU), le transfert d'appel sur occupation (CBL) et le rappel automatique (ACB). Le système a été décrit en utilisant le langage SDL [10], aussi bien pour la partie traitement de l'appel, que pour l'invocation de services et la gestion de l'utilisateur [4]. On se situe au niveau du plan fonctionnel global (GFP) en prenant en compte certains aspects du niveau plan fonctionnel distribué (DFP). Ce système est composé par plusieurs entités fonctionnelles, qui sont représentées par le bloc Network. Ce bloc comporte deux blocs: le bloc Basic Service, qui représente le service téléphonique de base (BCS) et le bloc Features (Fb) qui caractérise les services. Le bloc BCS contient trois processus: le Call Manager (qui s'occupe du côté gestion de l'appel), le Call Handler (qui prend en charge l'appel lui-même) et le bloc Feature Handler (qui permet l'accès aux services). Le bloc FB comprend cinq processus qui représentent les services: la liste noire, qui est instanciée deux fois afin d'obtenir une liste noire au départ, le service OCS, et une liste noire à l'arrivée TCS. Les autres services sont CFU, CBL, ACB. Ce bloc contient également un processus Feature Manager (qui établit un lien entre le Feature Handler et les services). L'architecture de cette spécification est décrite dans la figure 2.

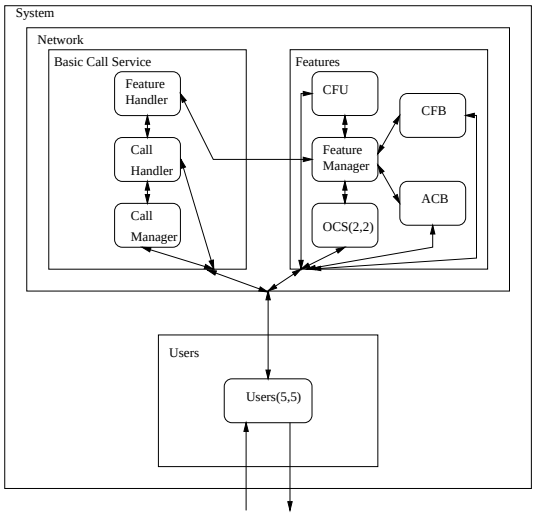

FIG. 2 - Architecture globale

Le modèle est décrit de telle sorte, qu'il permet l'exécution de différents appels en parallèle

et aussi les appels initiés par le réseau. L'environnement envoie des messages au processus Users, qui est modélisé par des instances de processus SDL qui composent le bloc Users. Le processus utilisateur (users) représente la combinaison de la ligne téléphonique, du terminal et de l'utilisateur. C'est relativement complet eu égard le cycle de vie du service, avec la modélisation des activations des utilisateurs, les désactivations, les mises à jour, les invocations. Afin de fournir une idée générale de la complexité de la spécification du système SDL, on présente dans la figure 2 l'architecture globale du système et dans la figure 3 quelques chiffres significatifs sur le système global. Ce dernier a été simulé en utilisant le mode de simulation exhaustive, afin d'obtenir le graphe d'accéssibilité complet. La figure 4 donne quelques informations sur le nombre d'états, de transitions, etc., tous obtenus après un arrêt manuel de recherche exhaustive. Il est impossible de construire le graphe d'accessibilité complet, ceci étant dû aux énormes ressources mémoires requises.

Lignes	3 098
Blocs	4
Processus	9
Procédures	12
Etats	88
Signaux	50
Macro définition	12
Timers	0

FIG. 3 - Mesures de la spécification de services

Nombre d'états	674 814
Nombre de transitions	2 878 800
Profondeur maximale atteinte	28
Durée	43mn 49s
Couverture des transitions	46.07%
Couverture d'états	70.37%

FIG. 4 - Simulation partielle de la spécification complète

On détaille ici les résultats obtenus pour la génération de tests pour les modules OCS et CFU. Ils sont des composants systèmes qui sont imbriqués dans le bloc Feature et qui ne possèdent aucun accès direct à l'environnement. Pour le test imbriqué du module OCS, on veut traverser au moins une fois chacune des transitions, qui sont décrites dans la figure 5 qui fournit la spécification en SDL du module. Les conditions d'arrêt ont été utilisées pour distinguer chaque transition du composant, l'écriture de celles-ci constitue la seule partie non automatisée de notre outil. La figure 6 illustre les conditions d'arrêt du module OCS, décrit à la figure 5.

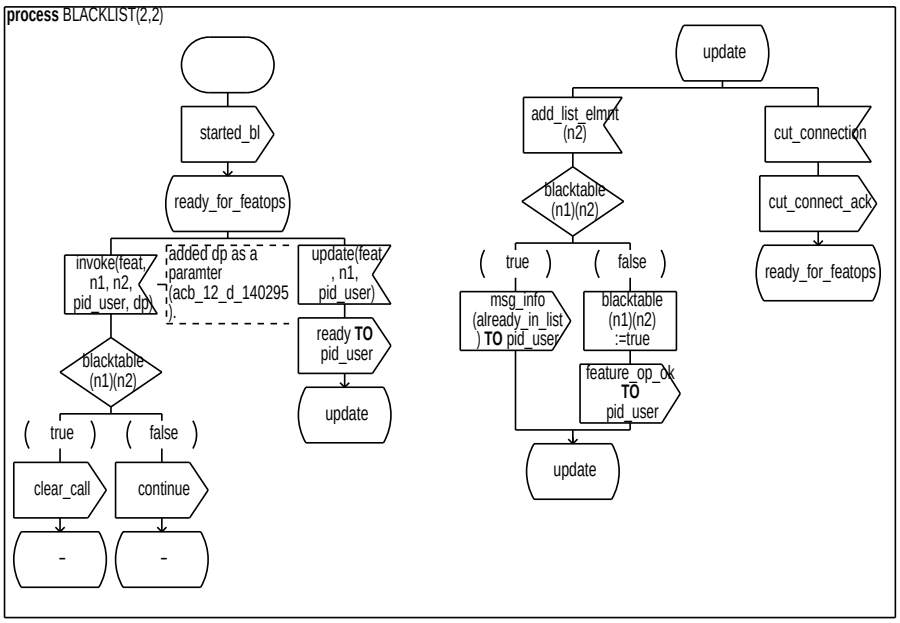

FIG. 5 - Processus liste noire

Afin d'exécuter la simulation du système, on le configure avec un fichier de démarrage, qui initialise certaines variables, telles que les services, les abonnés qui invoquent les services, les actions que chaque abonné peut réaliser (i.e. le nombre de raccrochages, le nombre d'activations de services, le nombre de désactivations, le nombre d'appels normaux). Pour cette étude de cas et les résultats obtenus, on a fixé ces variables à environ quatre-vingts actions par utilisateur.

stop if	output continue from blacklist		1
or	output clear_call from blacklist		#2
or	output ready from blacklist		#3
or	and	input add_list_elmnt to blacklist output feature_op_ok from blacklist	#4
or	trans blacklist : from_update_input_cut_connection		#5
or	and	input add_list_elmnt to blacklist output msg_info from blacklist	#6

FIG. 6 - Conditions d'arrêt du processus liste noire

Les résultats sont détaillés dans la figure 7. La ligne «Number of transitions» indique le nombre de transitions exécutées à chaque simulation (i.e. la taille de l'automate déployé). De fait il s'agit d'une mesure des ressources consommées, et non de la taille de la sous-séquence de test correspondante, qui est un chemin dans l'automate déployé. On remarquera que dans le pire des cas, pour trouver la condition d'arrêt input add_list_elmnt to blacklist and output msg_info from blacklist (stop #6), le simulateur passe seulement par 103 transitions. On voit clairement que l'algorithme Hit-or-Jump trouve effectivement les transitions non testées sans construire le graphe d'accessibilité complet.

En outre la séquence de test finale est courte. On notera que le temps fournit dans la figure ci-dessous correspond au temps CPU utilisateur réel (sur une Sun Sparc Ultra 1).

Conditions d'arrêt	#1	#2	#3	#4	#5	#6
Nombre de transitions	11	65	95	3	4	103
Profondeur atteinte	11	50	50	3	4	50
Durée (secondes)	2.5	4.4	6.7	8.6	10.4	11.1

FIG. 7 - Résultats de la simulation pour chaque condition d'arrêt

Une fois que toutes les transitions du composant imbriqué OCS sont traversées, on obtient une séquence de test unique, qui correspond à la longueur totale du chemin qui a été traversé de l'environnement à la dernière transition du module. La séquence obtenue a une longueur de 150 transitions; on a besoin seulement de 150 transitions pour couvrir tout le module OCS dans son contexte. Pour pouvoir comparer, on a également implanté l'algorithme de recherche aléatoire et on a obtenu une séquence de longueur 1402. Cela montre clairement que Hit-or-Jump produit une séquence de test avec la même couverture de fautes que la recherche aléatoire, mais d'une longueur nettement inférieure. On a également expérimenté cet algorithme sur le module CFU. La figure 8 illustre les résultats obtenus pour OCS et CFU, en terme de longueur de séquence de test. De plus, on a appliqué Hit-or-Jump sur le processus Responder du protocole INRES [7]. Les résultats pour le module Responder sont significatifs : on obtient une séquence de test de longueur 44 en mode BFS. On a également obtenu des séquences de test de longueur très différentes selon le mode de recherche avec l'algorithme Hit-or-jump.

Modules	OCS				CFU			
Modes	DFS	BFS	B-DFS	RW	DFS	BFS	B-DFS	RW
Profondeurs	50	50	50	-	100	100	100	-
#Stops	6	6	6	6	6	6	6	6
Sequences	834	150	167	1402	deadlock	137	261	586
#Sauts	18	1	0	-	< 70	0	0	-

FIG. 8 - Résultats pour les modules OCS et CFU

6. Conclusion

Nous avons présenté dans cet article un nouvel algorithme Hit-or-Jump pour l'exécution de test de composants qui sont imbriqués dans un système de communication complexe. Il s'agit d'une généralisation et d'une unification des recherches aléatoires et des recherches aléatoires guidées, mais aussi de l'algorithme structuré. De plus il ne donne pas lieu à l'explosion du nombre d'états, telle qu'elle se produit dans les recherches aléatoires.

Hit-or-Jump est une nouvelle technique pour la recherche dans l'espace des états du système. Nous l'avons appliqué pour le test imbriqué. Il peut également être utilisé pour la vérification et la validation, qui dépendent de la recherche dans l'espace des états du système, et notre méthode peut aider à traiter le problème de l'explosion du nombre d'etats.

Pour simplifier nous avons présenté une version directe de Hit-or-Jump. Il possède un nombre important de variations et de généralisations, et leurs implantations nécessitent de simples modifications sur la version présentée. Nous en décrivons briévement quelques unes. Pour un saut (Jump), on choisit de façon aléatoire et uniforme une feuille du graphe de recherche locale (arbre) et on continue à partir de là. On peut plutôt renforcer les priorités comme dans une recherche aléatoire guidée. Une autre variation est : s'il n'y a pas eu de touché (hit) pour un nombre important de sauts, on peut revenir sur un touché précédent, et ainsi sauter sur un nœud différent et continuer le test. L'idée sous-jacente est : rebrousser chemin quand on est «égaré». Bien que dans notre expérience avec l'architecture IN, nous n'ayons pas rencontré un tel problème, il peut ne pas être surprenant, dans le cas de test de composants imbriqués dans un système complexe. Aussi, lorsque l'on construit un arbre de recherche au fur et à mesure, on peut compresser les transitions internes de $C \times A$ [13] pour avoir plus d'espace.

Nous nous sommes concentrés sur la couverture de toutes les transitions de A. L'algorithme peut facilement être étendu à : (A) Couvrir quelques (pas nécessairement toutes) transitions de A, qui sont spécifiées par les utilisateurs ou les testeurs; (B) Couvrir quelques transitions et états de A avec des valeurs de variables spécifiées telles que des valeurs limites. On peut affecter une couleur distincte à chaque entité à couvrir, et lancer Hit-or-Jump jusqu'à ce que toutes les couleurs soient couvertes. Plusieurs approches [16, 15], utilisent les modèles de fautes. Nous avons une procédure générale, indépendante des modèles de fautes. Cependant, on peut affecter des couleurs aux entités de la machine composant sous test pour la couverture à partir de modèles de fautes, et notre procédure peut être utilisée pour la génération de test associée aux modèles de fautes.

Nous n'avons pas spécifié la profondeur de la recherche locale pour un saut, dans le cas où il n'y a pas de touché. Pour l'architecture IN, nous avons testé sur quelques valeurs de profondeur, i.e. 50 et 100. Intuitivement, une valeur de profondeur plus grande augmente la probabilité de toucher une partie non couverte du composant sous test. Cependant cela nécessite plus d'espace et de temps à chaque étape. En outre un saut trop long implique une sous séquence plus longue dans le test pour cette étape. On pense que le choix d'une bonne valeur de profondeur dépend du système sous test. Comme on l'a mentionné plus haut, on peut toujours choisir une valeur de profondeur comprise dans la limte de l'espace mémoire alloué. Comme recherche locale pour le Hit-or-Jump, nous avons testé à la fois la recherche en profondeur d'abord, en largueur d'abord, en prodondeur d'abord bornée. La recherche en largeur d'abord semble donner de meilleurs résultats dans le cas étudié.

7. BIBLIOGRAPHIE

[1] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. In *IWTCS'98*, Tomsk, Russia, Août 1998.

[2] E. Brinksma. A theory for the derivation of tests. In *Proc. IFIP WG6.1 8th Int. Symp. on Protocol Specification, Testing and Verification*. North-Holland, 1988.

[3] A. Cavalli, D. Lee, C. Rinderknecht, and Fatiha Zaidi. Hit-or-jump: An algorithm for embedded testing with applications to in services. In *Tech. Memo, Bell Laboratories*, Mai 1999.

[4] P. Combes and B. Renard. Service validation, tutorial. In *SDL Forum'97*, France, 1997.

[5] M. A. Fecko, U. Uyar, A. S. Sethi, and P. Amer. Issues in conformance testing: Multiple semicontrollable interfaces. In *Proceedings of FORTE/PSTV'98*, Paris, France, Novembre 1998.

[6] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV, LINCS 1102*, USA, Juillet 1996.

[7] D. Hogrefe. Osi formal specification case study: the inres protocol and service, revised. Technical report, Institut für Informatik Universität Bern, mai 1992.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[9] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework, International Standard IS-9646*, 1991.

[10] ITU. *Recommendation Z.100: CCITT Specification and Description Language (SDL)*, 1992.

[11] D. Lee, K. Sabnani, D. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach. In *IEEE Transactions on Communications*, volume 44, No.5, Mai 1996.

[12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84(8):1090–1123, Août 1996.

[13] L. P. Lima and A. Cavalli. A pragmatic approach to generating test sequences for embedded systems. In *Proceedings of IWTCS'97*, Cheju Island, Korea, Septembre 1997.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[15] A. Petrenko, N. Yevtushenko, and G. V. Bochmann. Fault models for testing in context. In *Proceeding of FORTE/PSTV'96*, Kaiserslautern, Germany, Octobre 1996.

[16] A. Petrenko, N. Yevtushenko, and G. V. Bochmann. Testing faults in embedded components. In *Proceedings of IWTCS'97*, Cheju Island, Korea, Septembre 1997.

[17] A. A. Petri. *Kommunikation mit Automaten*. Ph. D. thesis, Universitat Bonn, 1962.

[18] N. Yevtushenko, A. Cavalli, and L. P. Lima. Test suite minimization for testing in context. In *IWTCS'98*, Tomsk, Russia, Août 1998.

[19] J. Zhu and S. T. Vuong. Evaluation of test coverage for embedded system testing. In *IWTCS'98*, Tomsk, Russia, Août 1998.