Test de composants de service et exécution de tests sur une plate-forme CORBA

Ana Cavalli — Bruno Defude
Christian Rinderknecht — Fatiha Zaïdi

Institut National des Télécommunications
9 rue Charles Fourier
F-91011 Evry Cedex
{Ana.Cavalli, Bruno.Defude, Christian.Rinderknecht, Fatiha.Zaidi}@int-evry.fr

RÉSUMÉ. *Dans cet article nous présentons une plate-forme de tests de services et son application à un service d'audioconférence. Cette plate-forme permet la validation d'un service à partir de ses composants et l'exécution des tests obtenus sur une implantation réalisée sur une plate-forme de type CORBA. Deux aspects sont mis en avant, le test de composants imbriqués du service et l'exécution de ces tests sur une plate-forme CORBA. La méthode utilisée pour les tests de composants imbriqués est nouvelle. Elle est basée sur une génération de graphes partiels et évite ainsi l'explosion combinatoire du nombre d'état du système global. L'exécution de tests sur CORBA est également une nouveauté, il existe très peu de travaux sur la validation et l'exécution de tests dans ces environnements.*

ABSTRACT. *This paper presents a service test platform and its application to an audioconference service. This platform allows the service validation from his components and the execution of the obtained tests on a CORBA platform. Two aspects are relevant: the test of embedded components and their execution. The testing method for embedded components is new: it is based on the generation of partial graphs and avoids the combinatorial explosion of number of states of the global system. The test execution on a CORBA platform is also new, there are a few works on validation and test execution on these environments.*

MOTS-CLÉS : *Test de composants, test imbriqué, génération de tests, exécution de tests, CORBA.*

KEYWORDS: *Components testing, embedded testing, test generation, test execution, CORBA*

1. Introduction

Les opérateurs télécoms sont confrontés à l'explosion des nouveaux services qui utilisent le réseau téléphonique classique mais combinent également la transmision de la voix avec la transmission des images et du son. La difficulté inhérente à ce type de services réside dans le fait que l'on doit intégrer ces services de manière rapide et fiable afin de répondre aux exigences des utilisateurs.

Le réseau intelligent (RI) a constitué une première tentative pour résoudre ce type de problèmes et par la suite d'autres architectures ont été proposées comme TINA pour résoudre de manière plus efficace la conception et l'implantation de nouveaux services. Ces derniers doivent répondre aux attentes des utilisateurs et avoir une fiabilité certaine. Ceci suppose qu'après la phase de conception, ils seront validés et testés afin de garantir que les implantations proposées par les fournisseurs de services répondent à leur cahier des charges. De plus, ces services seront définis, en raison des contraintes de modularité et de réutilisabilité, à partir de composants de base qui pourront par la suite être combinés et réutilisés.

Dans cet article, nous présentons une plate-forme de test de services et son illustration par un cas d'étude (un service d'audioconférence). L'architecture générale de cette plate-forme est décrite de la façon suivante. L'entrée à partir de laquelle nous travaillons est un modèle SDL du service que nous voulons tester. À partir de ce modèle SDL, nous générons une séquence de tests par l'algorithme Hit or Jump (étape 1). Cette étape est entièrement automatisée et la mise en oeuvre fait appel au simulateur d'ObjectGEODE. Cette séquence est construite à partir du modèle SDL et est donc exprimée sur des «objets» SDL. Les valeurs associées aux signaux échangés notamment sont représentées dans le langage de types de SDL. Pour aller vers l'exécution de tests sur une architecture CORBA, cette séquence doit être transformée pour être exprimée sur des objets CORBA et donc vers le langage de type IDL CORBA (rôle de l'étape 2). Enfin la séquence de tests CORBA est exécutée par notre testeur sur une implantation du service et produit les résultats de tests (étape 3).

L'apport de ce travail réside dans la définition des méthodes pour le test de composants de services. Au CFIP99 [ION 99], nous avions déjà présenté une méthode pour le test imbriqué, mais cette méthode était basée sur la génération du graphe global associé au système testé. La méthode présentée dans cet article est nouvelle, elle est basée sur une génération de graphes partiels et évite ainsi l'explosion combinatoire du nombre d'états du système global.

Nous présentons aussi dans ce travail l'exécution de tests sur une plate-forme distribuée CORBA. Nous présentons les principes permettant de tester une implantation CORBA et qui supposent la combinaison de techniques de test interactives avec des techniques d'observation, appelées également techniques

de test passif. Cette approche est novatrice puisque, à notre connaissance, il existe très peu de travaux sur la validation et l'exécution de tests pour CORBA.

Ce travail a été effectué au sein du projet CASTOR du RNRT [CAS]. Les spécifications du service audioconférence nous ont été fournies dans ce cadre par le CNET-France Télécom.

Cet article est organisé de la manière suivante. Dans la section 2 nous donnons un bref aperçu de l'algorithme utilisé pour la dérivation des tests pour les composants du service. Cette section inclut également une comparaison de notre méthode avec les méthodes existantes. La section 3 présente le cas d'étude, le service d'audioconférence et ses composants. De plus dans cette même section nous présentons les résultats des expériences que nous avons réalisées. Puis la section 4 présente une description de la plate-forme CORBA d'exécution de tests et explique les principes qui ont guidé sa conception. Enfin, la section 5 donne les conclusions de ce travail.

2. Méthode de génération de séquence de test pour les composants

2.1. *Préliminaires*

Dans cette section nous présentons brièvement la méthode utilisée pour la génération de tests de composants. Pour une description plus complète nous vous renvoyons à [CAV 99]. Pour mieux comprendre la méthode nous allons donner préalablement quelques définitions. Cette section inclut également une comparaison avec les méthodes et outils existants pour le test imbriqué (ou test des composants dans son contexte).

L'objectif de la méthode est de tester des composants dans leur contexte (qui est généralement le reste du système), alors qu'on n'a pas d'accès direct au composant. Le système complet est exprimé sous forme de CEFSM (*Communicating Extended Finite State Machine*), à savoir l'environnement ainsi que les composants sous test.

Une machine à états finis étendue (EFSM) est une machine d'états finis enrichie par des variables. Le franchissement d'une transition dépend de l'évaluation du prédicat, celui-ci doit être évalué à vrai. Une transition définit des actions sur les variables. Le service d'audioconférence que nous allons étudier est spécifié en utilisant le langage SDL [ITU92]. La spécification du comportement d'un processus en SDL est basée sur le modèle des EFSM. Contrôler la valeur de variables locales impose l'existence d'une condition (prédicat) pour passer d'un état à l'état suivant. Les actions associées à une transition incluent l'exécution de tâches, des appels de procédure, la création dynamique de processus (SDL contient les concepts de «type» et d'«instance de type») ainsi que l'activation et désactivation de temporisateurs.

2.2. *Description de l'algorithme Hit-or-Jump*

Pour le test des composants on utilise un algorithme qui permet de couvrir toutes les transitions (fonctionnalités) du composant dans son contexte. On construit à chaque étape un graphe d'accessibilité partiel, on évite ainsi le problème d'explosion du nombre d'états. L'algorithme permet de générer une séquence de test qui est un tour de transition du composant générique dans son contexte. L'algorithme Hit-or-Jump a été conçu pour éviter d'être piégé dans une partie du graphe, sans pour autant construire le graphe d'accessibilité complet. Il ne nécessite pas non plus la construction du graphe d'accessibilité du $Contexte \times Composant$, 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 (bornée ou non) d'abord ou largeur d'abord.

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 transitions de A sont couvertes.

Exécution. La séquence de test est initialement vide. Tant que des transitions du composant demeurent non trouvées, on lance une simulation exhaustive. Celle-ci peut effectuer la recherche selon plusieurs modes, en largeur d'abord, en profondeur d'abord ou en profondeur bornée. Deux cas peuvent se produire:

1. un Hit: on a trouvé une transition du composant;

2. un Jump: on s'est arrêté sur une limite, sans trouver de transition du composant.

Hit. On s'est arrêté sur une transition non couverte du composant. Puis:

1. on étend la séquence de test avec le chemin dans l'automate déployé, obtenu après arrêt de la simulation exhaustive, de la racine à la transition trouvée;

2. on joue la séquence de test ainsi obtenue contre la spécification; on obtient un nouveau vecteur d'état;

3. on élimine l'automate déployé;

4. s'il reste des transitions non couvertes, on lance une nouvelle simulation exhaustive, sinon on retourne la séquence.

Jump. On atteint une limite pré-définie par l'utilisateur (par exemple une profondeur limite). Alors:

1. on considère un arbre recouvrant de l'automate déployé;

2. on choisit aléatoirement un chemin dans cet arbre recouvrant, de l'état initial à une feuille, et on l'ajoute à la séquence de test;

3. on répète les étapes 2 et 3 du ***Hit***.

Cet algorithme évite de boucler sur le même état, dans la mesure où l'on choisit aléatoirement dans l'arbre recouvrant un chemin. En outre il traite également les cas de non déterminisme.
À noter que l'application de l'algorithme Hit-or-Jump suppose que la spécification est correcte dans le sens qu'elle ne présente pas de blocages à l'exécution, des états inatteignables, etc.

2.3. *Comparaison avec les méthodes et outils existants*

Il existe depuis plusieurs années des recherches autour de la définition de méthodes pour la génération automatisée des tests, et depuis seulement quelques années des recherches autour du test imbriqué [BOU 98, FEC 98, LEE 96, ZHU 98].

Dans ce travail nous ne prétendons pas faire une comparaison avec tous les travaux cités plus haut, qui présentent tous un apport théorique et pragmatique à la problématique qui nous intéresse. Parmi ces travaux, nous allons comparer notre travail avec ceux qui ont produit des outils permettant d'effectuer le test des composants, à savoir l'outil Test Composer [KER 99] et les travaux présentés dans [CLA 96] qui sont en partie intégrés dans Test Composer.

Le point commun à ces méthodes est l'idée d'interrompre la simulation exhaustive selon un certain critère. Dans [CLA 96] il était proposé d'appliquer successivement les trois heuristiques suivantes:

1- Borner le nombre d'états parcourus du module imbriqué, et relever le pourcentage de transitions trouvées...

Dans notre méthode, nous nous arrêtons sur une transition du module imbriqué;

2- ... ensuite, s'il reste des transitions à trouver, lancer une simulation exhaustive guidée par une métrique sur les états SDL (la proximité, par exemple)...

Dans notre méthode, nous relançons simplement la simulation;

3- ... et finalement s'il reste toujours des transitions, choisir et exécuter des chemins (déjà trouvés) proches (au sens de la métrique) des transitions restantes et de lancer une simulation exhaustive, comme au début.

En revanche, dans notre méthode, si nous ne trouvons rien nous faisons un saut aléatoire parmi ce qui a déjà été exploré. L'idée est donc différente, nous parions sur la non-proximité pour sortir d'éventuelles chausse-trapes.

De plus, définir une métrique est un procédé compliqué. Il faut définir des attributs mathématiques et mettre en correspondance l'attribut empirique (ici la notion de proximité) avec un calcul sur ces attributs formels. Ensuite, et c'est

très important, il faut établir un modèle statistique sur la répartition des valeurs de ces attributs formels par rapport à la «réalité» (ici la proximité empirique). Puis il faut se livrer à une campagne de mesures et évaluer empiriquement la pertinence de celles-ci par rapport au concept empirique. Si c'est le cas, on obtient alors une métrique, avec un certain modèle statistique. Sinon on change et on recommence. D'ailleurs rien n'assure que la métrique que l'on garde est la meilleure, c'est une méthode très expérimentale, et donc proche des faits, mais qui est très exigeante. Ensuite, d'un point de vue pratique, il est difficile de vraiment comparer car les deux dernières heuristiques (celles qui introduisent la métrique, justement) n'étaient pas implémentées, aux dires des auteurs. Et il faut mentionner egalement que ces métriques n'ont pas été implantées dans l'outil TVEDA.

Nous donnons dans ce qui suit un résumé de la réponse des auteurs de [CLA 96], aux remarques précédentes car nous considérons que ceci aidera à mieux comprendre les caractéristiques de chaque méthode:

«L'utilisation d'un saut aléatoire (*Jump* dans l'algorithme) présente comme avantages la simplicité de sa mise en œuvre et son efficacité par rapport à la définition des métriques. L'intérêt de la troisième heuristique proposée dans [CLA 96], était d'essayer de résoudre des problèmes de type, changement de comportement quand le compteur de retransmission est arrivé à 128; pour cela il faut arriver à faire passer le simulateur 127 fois par un échec de transmission pour enfin atteindre le cas 128. Un des auteurs avait écrit des métriques de ce type pour le LAPD, mais elle ne sont pas faciles à automatiser.»

À ce dernier commentaire, nous pouvons répondre que nous pouvons tester ce type de transitions. Il faudra pour cela mettre comme condition d'arrêt une condition sur le compteur de retransmission (par exemple, «le compteur atteint la valeur 128»): ceci fera passer le simulateur par les valeurs précédentes (éventuellement les 127) mais il s'arrêtera dès qu'on sera passé par la valeur 128.

Par rapport à l'outil Test Composer il est mentionné dans [KER 99] que cet outil est basé sur deux prototypes: TVEDA et TGV. TVEDA permet de dériver des objectifs de test, en utilisant des techniques présentés dans [CLA 96] et que nous avons déjà commentés dans les paragraphes précédents. TGV utilise des techniques de vérification basées sur le "model checking" afin de générer à partir des objectifs de test les cas de test pertinents. Dans notre cas, nous n'avons pas besoin d'un objectif de test externe comme une entrée de notre outil. Par rapport à cet aspect, notre outil est plus proche de TVEDA. En effet, nous pouvons générer les tests directement à partir de la spécification. Il peut s'agir de tester une transition ou plusieurs transitions, ou les transitions d'un module imbriqué (comme c'est le cas dans cet article).

3. Descriptif de la spécification

Le service d'audioconférence est spécifié en utilisant le langage SDL, comme nous l'avons déjà mentionné dans la section précédente. Ce service permet à un nombre déterminé d'utilisateurs d'établir une communication groupée. Un abonné, ou souscripteur à la conférence, joue un rôle particulier, le modérateur. Il a en charge la réservation du pont de conférence pour une heure et un jour donné. La conférence ne peut commencer avant l'arrivée du modérateur.

Cette spécification repose sur une architecture de réseau intelligent et sur le concept de composants réutilisables, les SIB (*Service Independent Building Blocks*). La recommandation Q.1201 définit le modèle conceptuel du réseau intelligent. Ce modèle est divisé en quatre plans — chaque plan étant une vue abstraite du système. La spécification traitée met davantage l'accent sur le plan fonctionnel global, qui est responsable de la création de service, et sur le plan des services. On s'intéresse plus particulièrement ici à la vue usager du réseau et des services. Ce plan modélise le réseau comme une entité unique, qui exécute les services.

Dans l'architecture du système décrite dans la figure 1 on trouve trois blocs principaux, le bloc *Users*, qui comprend trois processus, un processus *users*, un processus *conference*, un processus *term_ manager*, le bloc CAA (Commutateur à autonomie d'acheminement), et le bloc SCF (Service Control Function). Le processus *users* (seul processus à interagir avec l'environnement) correspond au comportement de l'utilisateur à savoir décrocher, raccrocher, composer un numéro, recevoir les différentes tonalités et parler, le comportement est défini à un niveau de granularité assez fin. Le processus *conference*, correspond au terminal pont de conférence, qui peut sonner, être décroché... Le processus *term_ manager* quant à lui gère les données SDL du bloc, ainsi que les créations des autres processus. Le bloc CAA assure la fonction de commutation; il est divisé en quatre processus, un processus *obcsm* pour traiter un demi-appel entrant, et un *tbcsm* pour le demi-appel sortant, et un processus CCF (Call Control Function) et un SSF (Service Switching Function). Les processus *tbcsm* et *obcsm* sont deux automates associés à un CCF qui permettent d'établir une communication entre deux utilisateurs (automate d'appel). Un automate est associé à la partie appelante et un autre à la partie appelé. Le processus CCF établit, manipule et relâche les appels. Le SSF, quant à lui, réalise la connexion en association avec un CCF, entre un utilisateur et l'entité de contrôle de service SCF. Dans la figure 1 nous ne donnons pas tous les détails de connexion entre les différents éléments par souci de visibilté de l'architecture. Tous ces éléments sont bien évidemment spécifiés en SDL. On ne traite que très partiellement les communications réseau, liées au protocole INAP, seulement quelques primitives sont spécifiées. Dans cette spécification l'accent est mis sur l'aspect modulaire de la conception de service, et son découpage en SIB. Un SIB est une brique de base qui comprend des paramètres d'instanciation en entrée et fournit un résultat.

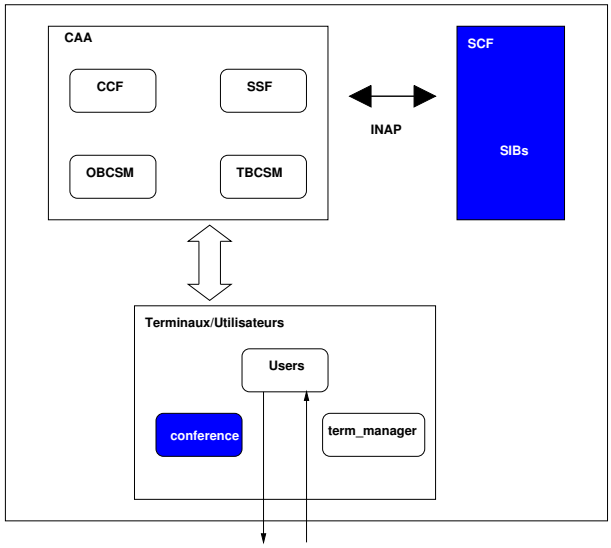

Figure 1. *Architecture de la spécification*

3.1. *Génération de tests pour les composants du service*

La brique de base est un composant monolithique, réutilisable et normalisé servant à construire des services. Le SIB prend en entrée des paramètres statiques les SSD (*Service Support Data*) et les CID (*Call Instance Data*), il peut être activé à un point d'activation POI (*Point of Initiation*) et peut avoir une ou plusieurs terminaisons POR (*Point of Return*). Les paramètres dans l'application qui nous intéresse ici, les SSD sont liés aux accès à la base de données, ils permettent également de préciser si le SIB doit avoir un comportement synchrone ou asynchrone. Au niveau de l'interface du SIB pour les paramètres de sorties on fait apparaître les champs qui ont été valués suite à l'exécution du SIB

Les SIB du réseau intelligent sont conçus de manière procédurale. À chaque composant correspond une procédure SDL. Aussi pour assurer le service, on utilise un processus qui a en charge d'assurer l'enchaînement des briques de base (cet enchaînement est séquentiel). Ce processus joue le rôle de «colle» entre les composants.

Résultats obtenus

Le service d'audioconférence est assuré par un enchaînement particulier de ces briques de base. Les différents composants sont génériques, ils modélisent chacun une fonction. Afin de constituer une logique de service exécutable, ils doivent être instanciés par des paramètres d'instanciation. Ces paramètres per-

mettent de définir différents comportements d'un composant. Les différents composants de service sont les suivants:

– DCL assure le déclenchement de l'appel;

– EDA permet l'établissement de l'appel;

– MVl_ann délivre les messages vocaux aux utilisateurs;

– MVL_exp a en charge les messages vocaux d'exploitation;

– RSC_incr, RSC_decr, RSC_obs gèrent respectivement l'incrémentation de la ressource «pont de conférence», la décrémentation et l'observation.

Lignes	9873
Blocs	3
Processus	13
Procédures	49
États	307
Signaux	67
Macro définition	29
Temporisateurs	1

Figure 2. *La spécification en chiffres*

DCL	100%	32
EDA	65%	119
MVL_ann	50%	55
MVL_exp	80%	197
RSC_incr	100%	36
RSC_decr	100%	131
RSC_obs	80%	140

Figure 3. *Résultats des composants*

La figure 2 présente quelques informations sur la complexité de la spécification du service traité (nombre de lignes de code, des blocs, des processus, etc). La figure 3 présente les résultats obtenus. La première colonne indique les taux de couverture des interactions de chaque composant imbriqué et la deuxième donne la longueur de la séquence obtenue.

Le module «pont de conférence».

Nous avons appliqué également l'algorithme de génération de séquence de test au module «pont de conférence». Il nous paraissait plus judicieux de montrer la séquence obtenue pour ce module, dans la mesure où elle permet de bien comprendre le fonctionnement de réservation d'un pont de conférence, les séquences pour les composants étant moins explicites. Nous avons obtenu une séquence de longueur 247 présentée dans le format MSC [ITU96] et dans le format TTCN [ISO91].

La figure 4 est le MSC simplifié de la séquence de test obtenue pour le module représentant le terminal «pont de conférence». Cette séquence met en évidence toutes les interactions de ce module, en fait elle permet de couvrir toutes ses transitions. On ne montre ici que les échanges de signaux entre les deux entités, à savoir le commutateur (SSF) et l'utilisateur, avec lesquelles le «pont de conférence» interagit. L'environnement ici est schématisé par l'encadrement de la figure. Les scénarii que l'on retrouve dans les échanges de signaux

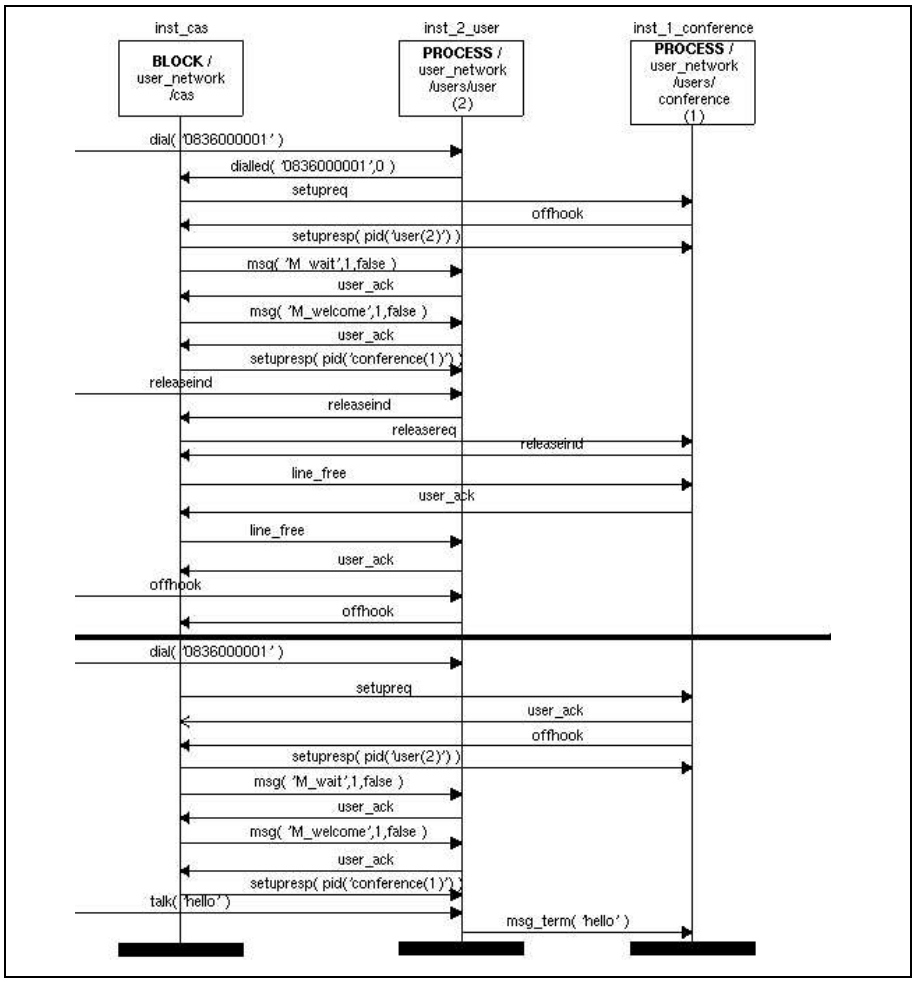

Figure 4. *Séquence de test simplifiée du module «pont de conférence»*

de la figure 4 sont les suivants :

– L'utilisateur 2 fait une demande de connexion à l'audioconférence en appelant le numéro '083600001'. Au niveau de la conférence cela se traduit par la réception d'un signal setupreq. Le module conférence envoie un signal offhook au commutateur. Ce dernier, suite à un enchaînement d'échanges avec le PCS et d'activation des différents composants, envoie un signal setupresp qui signifie que la communication peut s'établir. Dans un premier temps l'utilisateur reçoit un message de mise en attente, et dès que la communication est établie un message d'accueil. Dès réception de ce dernier l'utilisateur décide de rac-

crocher il envoie un signal releaseind au commutateur, celui le retransmet au module conférence par le signal releasereq. Ce dernier acquitte la demande de déconnexion par le signal releaseind, ensuite il reçoit un message line_free qui indique que la ligne est de nouveau libre.

– le deuxième partie de la figure met en évidence le scénario suivant : L'utilisateur 2 appelle le pont, il reçoit le message d'attente puis la conférence est ouverte; les abonnés en attente reçoivent le message d'accueil, suite à cela la communication est établie, ils peuvent enfin parler, cet échange se traduit par la réception d'un signal msg_term.

On fournit également une partie du TTCN obtenu. La séquence obtenue dans ce format exprime la séquence de test en test boîte noire, les points d'observation et de contrôle sont situés au niveau système. On voit donc les échanges entre l'environnement et le bloc utilisateur.

Test Case Name : network_RI_0
Group : network_RI/
Purpose :
Default : DEF_0
Comments : Generated by test oriented simulation of the test purpose network_RI.msc

Nr	Behaviour Description	Constraints Ref	Verdict
1	env2users!offhook output		
2	env2users!composer START TAC	composer	
3	users?msg CANCEL TAC	msg_3	
4	env2users!composer	composer_4	
5	env2users!releaseind		
6	env2users!offhook		
7	env2users!composer START TAC	composer_2	
8	env2users?msg CANCEL TAC	msg_3	
9	env2users!composer	composer_4	
10	env2users!talk	talk_6	
11	env2users!offhook		
12	env2users!composer START TAC	composer_7	
13	env2users?msg CANCEL TAC	msg_3	
14	env2users!composer START TAC	composer_7	
15	env2users?msg CANCEL TAC	msg_3	(P)
16	? TIMEOUT TAC		F
17	? TIMEOUT TAC		F
18	? TIMEOUT TAC		F
19	? TIMEOUT TAC		F

4. L'exécution des séquences de tests sur CORBA

4.1. *Problématique*

L'architecture cible d'exécution que nous avons choisie est CORBA. CORBA est aujourd'hui une solution industriellement reconnue notamment dans le

monde des télécommunications et les opérateurs et équipementiers du secteur ont considérablement investi dans cette technologie. On peut donc penser que les futurs services de télécommunications seront basés sur cette solution.
CORBA est une architecture d'interconnexion de logiciels hétérogènes. Pour réaliser l'interconnexion, CORBA utilise la notion d'objets et d'invocations d'opérations sur les objets et offre une abstraction de bus logiciel pour transporter les invocations appelée ORB (*Object Request Broker*). Chaque logiciel à interconnecter est décrit comme un ensemble d'objets (au moins un). Pour supporter l'hétérogénéité, une description de chaque logiciel doit être produite dans un langage qui va servir de pivot et qui s'appelle IDL CORBA (Interface Definition Language). En suivant les architectures de test proposées pour les protocoles classiques, telles qu'elles sont décrites dans le norme ISO 9646, nous avons considéré qu'une architecture de test adaptée aux environnements comme CORBA devait comporter deux types de composants, des composants testeurs et des composants sous test. Ceux-ci peuvent être répartis sur différents sites mais nous avons choisi d'avoir un seul composant testeur (donc centralisé).
Une première version d'architecture de test pour CORBA a déjà été développée dans notre groupe [LIM 97]. Cette version suppose que les séquences de tests et le comportement des composants sous test sont décrits sous forme d'automates à états finis. Pour lever cette limitation, nous avons développé une nouvelle version du testeur. Ce nouveau testeur doit permettre de tester n'importe quel composant et notamment de tester des composants dont on ne possède pas le code source de l'implantation.
Nous avons également fait le choix d'être indépendant des vendeurs de technologie CORBA de manière à ce que notre testeur soit portable sur tout environnement CORBA. Nous verrons ci-après que ce choix nous a contraint par rapport aux possibilités d'observation des invocations d'opérations entre objets. Une autre solution serait de construire un ORB CORBA dédié au test que nous pourrions alors adapter à loisir notamment pour faire de l'observation. Cette solution en plus de la non portabilité est coûteuse en développement et c'est pourquoi nous ne l'avons pas choisi. D'autres ont choisi cette voie pour augmenter les capacités d'observation et de réactions d'un ORB [GRA 99] en modifiant le code source de l'ORB Orbacus.
Nous avons choisi de développer nos applications CORBA en utilisant le langage Java. Cela doit nous permettre de prototyper assez rapidement des interfaces utilisateurs conviviales (par exemple pour consulter les résultats de tests). Nous utilisons deux ORB différents pour vérifier la portabilité (Orbix de IONA Tech. et Orbacus de OOC).

4.2. *Entrées du testeur*

L'algorithme Hit-or-Jump génère des séquences de tests à la fois actives et passives. Dans ce type de test, le testeur doit être capable de stimuler le composant à tester et comparer le résultat obtenu au résultat attendu, mais

aussi d'observer les messages (invocations d'opérations pour CORBA) échangés entre les composants testés.

Une séquence de tests comprend les entrées suivantes :

– des interactions vers le composant à tester. Cette entrée est décrite comme un quadruplet de messages, le premier sert à initialiser le composant à tester, le deuxième correspond à l'entrée, le troisième permet de récupérer le résultat de l'entrée (l'entrée modifie des états que ce message vient interroger) et le quatrième repositionne le composant dans un certain état. Chaque message comprend un nom, un objet destinataire et des valeurs d'arguments;

– des messages à observer (nom du message, objet destination et valeurs des arguments). Il faut noter que l'ordre des messages a de l'importance.

4.3. *Architecture du testeur*

Fonctionnellement le testeur comprend deux parties (voir figure 5). La première sert à exécuter les messages CORBA correspondant aux entrées de type interactions et l'autre est un observateur qui permet de traiter les entrées de type messages à observer. Du point de vue CORBA, la première partie est un objet CORBA client générique qui est capable d'invoquer n'importe quelle opération sur n'importe quel objet. Le service de nommage est utilisé pour pouvoir résoudre les noms utilisés dans la séquence de tests en adresses d'objet CORBA (appelées IOR dans la terminologie CORBA). Le client est réalisé grâce à l'interface d'invocation dynamique CORBA qui permet de construire dynamiquement des invocations en consultant le dépôt d'interfaces IDL.
Un des problèmes à résoudre ici est le typage des résultats. En effet il faut pouvoir construire dynamiquement des objets bien typés pour récupérer les résultats des invocations. Cela suppose que le programme connaisse soit statiquement tous les types, soit il doit construire dynamiquement les types Java manquant (c'est ce que nous avons fait).
Les composants testeurs sont équipés de chronométreurs afin de détecter les blocages et les boucles vivaces [1]. Une valeur de temporisateur différente est attribuée à chaque transition que nous envisageons de tester. En pratique, on donne une valeur de temporisateur par défaut à toutes les transitions (certaines exigences de qualité de service, i.e des exigences de performance, peuvent être exprimées en utilisant certaines valeurs de temporisation pour des transitions particulières).
La deuxième partie est plus délicate puisqu'elle nécessite de devoir observer des invocations entre objets. De plus les ORBs n'offrent pas actuellement de moyens d'observer les invocations d'objets. Après avoir étudié les mécanismes de type intercepteurs qui sont en cours de standardisation à l'OMG et pour lesquels quelques implantations sont disponibles, nous nous sommes rapidement

1. *livelocks*

aperçus qu'ils nécessitaient de modifier le code source et donc que ce n'est pas une solution viable.
Finalement nous avons opté pour une solution basée sur l'observation du trafic IIOP. IIOP (*Internet Inter Orb Protocol*) est l'implantation au dessus de TCP du protocole GIOP (*Generic InterOrb Protocol*) qui supporte l'invocation d'objets entre différents ORB. Toute l'information qui nous intéresse se trouve dans les trames IIOP sous forme non exploitable directement. Cette solution a l'avantage d'être indépendante des ORB et surtout de ne pas nécessiter de modification ou recompilation des composants testés. L'inconvénient est que pour des raisons d'optimisation, les ORB n'utilisent pas IIOP pour les appels locaux. Ceci nous oblige à forcer la répartition des composants que nous devons observer pour que leurs interactions se fassent forcément sous forme de trafic IIOP. L'architecture logicielle de l'observateur est décrite figure 5.
Puisque nous sommes dans un contexte réparti nous avons besoin d'avoir un observateur par machine sur laquelle est déployée l'application testée. Ces observateurs doivent coopérer entre eux pour construire une histoire globale des invocations entre objets (la coopération est facilitée puisque les invocations sont bloquantes). L'observateur est donc une application répartie que nous avons choisi de développer selon une architecture CORBA. On dispose d'un objet CORBA qui est l'observateur global et qui va interagir avec le testeur via une interface IDL (ce qui fait qu'ils ne sont pas nécessairement sur la même machine). Cet observateur global va construire une histoire global des invocations entre objets CORBA en collectant les informations auprès des observateurs locaux. Le dialogue entre l'observateur global et les observateurs locaux se fait également via une interface IDL (dans les deux sens d'ailleurs). On peut remarquer que le testeur est maintenant réparti.

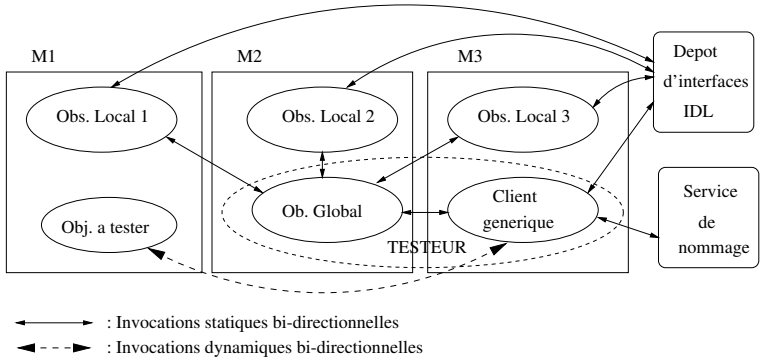

Figure 5. *Architecture du testeur*

Un observateur local doit observer le trafic IIOP et retrouver l'information qui nous intéresse dans les trames échangées. Nous voulons retrouver le nom des opérations invoquées, le nom de l'objet destinataire et les valeurs des arguments de l'opération. Le problème est que le codage dans une trame IIOP

est orienté transport et donc tout se trouve représenté sous forme d'une chaine d'octets (on dispose juste d'informations type version du protocole, type de message IIOP et nom de l'opération). Pour décoder la chaine d'octets nous devons d'une part disposer du service de nommage pour retrouver les objets destinataires et d'autre part du dépôt d'interface pour nous permettre de retrouver les types des arguments. A partir de ces informations, on peut décoder le contenu de la trame.
Le client générique est opérationnel par contre l'observateur est en cours de développement. Les interfaces IDL CORBA sont décrites et un premier niveau d'observation est réalisé en récupérant les trames IIOP au-dessus de l'analyseur de protocole TCPDump. Il nous reste à implanter l'observateur global et surtout à utiliser le dépôt d'interfaces pour analyser complètement les trames IIOP.

5. Conclusion

Dans cet article nous avons présenté d'une part une architecture globale de test pour des services répartis comprenant la génération de séquences de test de composants d'un service et d'autre part l'exécution des séquences de tests produites sur une implantation CORBA du service. Notre approche a été validée sur un service d'audioconférence réel.

Le service complet a été décrit en utilisant le langage SDL, l'application de l'algorithme Hit-or-Jump n'a mis au jour aucun blocage. Pour présenter les résultats de nos travaux, nous avons sélectionné un composant du service d'audioconférence qui est au cœur du service, le procesus «pont de conférence», qui coordonne tous les autres composants et illustre de manière claire ce qu'on imagine être un service d'audioconférence. Les autres composants étaient des composants génériques qui peuvent être présents dans d'autres services de télécommunication et pour lesquels nous avons également généré les tests. Nous avons produit les tests de ce composant dans son contexte, et nous l'avons traduit dans le format TTCN et MSC.

Nous avons défini une architecture pour le testeur qui combine une partie active (basée sur la stimulation de l'implantation) et une partie passive (basée sur l'observation des échanges entre les objets testés). Cette dernière est en cours de développement.

Cependant les résultats obtenus montrent que l'utilisation des méthodes formelles facilite considérablement la tâche des concepteurs et des développeurs de services et qu'elles sont utilisables pour des services réels. Dès la phase de spécification jusqu'à la phase d'implantation et de test, nous avons utilisé des techniques formelles de description (le langage SDL) et des méthodes formelles de test. De plus, nous avons montré qu'il est possible de tester les composants d'un service dans le contexte d'autres composants. Cela représente une avan-

cée très importante vers la validation des services et vers la conception des composants de service réutilisables.

6. Bibliographie

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

[CAS] CASTOR, « Conception d'un Atelier de création de Services pour les plateformes Tina, CORBA et RI. », Partenaires Alcatel, France-Telecom-CNET, Verilog,INT. Projet pré-compétitif RNRT, `www-inf.int-evry.fr/~castor`.

[CAV 99] Cavalli A., Lee D., Rinderknecht C., Zaïdi F., « Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services », *Proceedings of FORTE/PSTV'99*, Beijing, China, Octobre 1999.

[CLA 96] Clatin M., Groz R., Phalippou M., Thummel R., « Two approaches linking test generation with verification techniques », Cavalli A., Budkowski S., Eds., *Protocol Test Systems VII*, Chapman & Hall, 1996.

[FEC 98] Fecko M. A., Uyar U., Sethi A. S., Amer P., « Issues in conformance Testing: Multiple Semicontrollable Interfaces », *Proceedings of FORTE/PSTV'98*, Paris, France, November 1998.

[GRA 99] Gransart C., Merle P., Geib J., « GoodeWatch: Supervision of CORBA Applications », *Proceedings of ECOOP'99 Workshop on Object-Orientation and Operating Systems*, Lisbonne, Portugal, Juin 1999.

[ION 99] Ionescu M., Cavalli A., « Test imbriqué du protocole MAP-GSM », *Proceedings of CFIP'99*, Nancy, Avril 1999.

[ISO91] ISO, « Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework, International Standard IS-9646 », 1991.

[ITU92] ITU, « Recommendation Z.100 : CCITT Specification and Description Language (SDL) », 1992.

[ITU96] ITU, Geneva, « Rec. Z. 120 Message Sequence Charts, (MSC) », 1996.

[KER 99] Kerbrat A., Jeron T., Groz R., « Automated test generation from SDL specifications », Dssouli R., Bochman G., Lahav Y., Eds., *SDL'99*, Elsiever Science, 1999.

[LEE 96] Lee D., Sabnani K., Kristol D., Paul S., « Conformance Testing of Protocols Specified as Communicating Finite State Machines - A Guided Random Walk Based Approach », *IEEE Transactions on Communications*, vol. 44, No.5, May 1996.

[LIM 97] Lima L. P., Cavalli A., « Exécution de tests de services sur une plateforme distribuée », *Actes NOTERE'97*, Pau, France, Novembre 1997.

[ZHU 98] Zhu J., Vuong S. T., « Evaluation of test coverage for embedded system testing », *IWTCS'98*, Tomsk, Russia, August 1998.