8e Conférence Internationale de MOdélisation et SIMulation - MOSIM'10 - 10 au 12 mai 2010 - Hammamet - Tunisie
« Évaluation et optimisation des systèmes innovants de production de biens et de services »

Robustesse des services web persistants *

Sébastien Salva
LIMOS - UMR CNRS 6158
Université d'Auvergne
Campus des Cézeaux, Aubière, France
salva@iut.u-clermont1.fr

Issam Rabhi
LIMOS - UMR CNRS 6158
Université Blaise Pascal
Campus des Cézeaux, Aubière, France
rissam@isima.fr

RÉSUMÉ : Les services web font partis des techniques de développement dites émergentes et sont de plus en plus utilisés aujourd'hui pour construire des applications. De part leurs natures, ces web services doivent être fiables et robustes. C'est pourquoi nous proposons, dans cet article, une méthode de test de robustesse de services web persistants, modélisés par des spécifications symboliques. Une partie de cet article est réservée à l'analyse de la robustesse en se focalisant sur l'environnement SOAP, qui réduit le contrôle et l'observation des messages transmis. Nous en déduisons notamment les aléas qui sont pertinents pour le test et ceux qui sont ignorés et bloqués par les processeurs SOAP. Nous générons les cas de test en complétant la spécification et en injectant, dans les cas de test, des valeurs inhabituelles prédéfinies.
MOTS-CLÉS : test de robustesse, services web persistants, STS, architecture de test.

1 Introduction

Les architectures SOA et en particulier les services web sont de plus en plus employés dans les entreprises pour construire des applications orientées Business ou Web. Ces composants distribués peuvent être utilisés, par exemple, pour assurer l'interopérabilité entre des applications hétérogènes, pour extérioriser du code fonctionnel d'une manière standard, ou pour composer des services (orchestration ou chorégraphie). Cette interopérabilité est garantie grâce à un ensemble de standards définis par les consortiums W3C et WS-I. Notamment, le profil WS-I basic réunit le protocole SOAP, qui permet d'invoquer un service web via XML, et le langage WSDL, qui permet de décrire leurs interfaces.

Les services web et protocoles associés correspondent souvent aux briques de fondations d'applications vastes et complexes. Afin de fiabiliser de telles applications, des processus de qualité et de certification logiciel, comme le CMMI (*Capability Maturity Model Integration*) sont appliqués par de nombreuses sociétés de développement. Ces processus de qualité sont notamment composés d'un ensemble d'activités de tests, comme le test de conformité ou de robustesse, qui correspond au sujet de cet article. La robustesse des services web est primordiale: ceux-ci sont distribués par nature et peuvent être appelés par des applications clientes très différentes. C'est pourquoi ils doivent aussi se comporter correctement en cas de réception d'événements non spécifiés, appelés des *aléas*.

Nous proposons d'étudier dans cet article la robustesse des services web persistants. Un tel service est gardé en mémoire à travers une session et possède un état interne qui évolue au fils des séquences d'appels. Nous représentons ces services web comme des boîtes noires à partir desquelles seuls les messages SOAP (requêtes et réponses) sont observables. De ce fait, nous modélisons l'état d'un service web par une spécification symbolique qui représente les opérations qui sont appelées ainsi que les paramètres associés. Le challenge apporté par le test des services web persistants concerne l'environnement SOAP dans lequel ceux-ci "baignent". Celui-ci réduit le contrôle et l'observation des messages émis par et vers le client. Ainsi, n'importe quel message habituellement observable, comme une réponse classique, est encapsulé puis émis vers le client via le protocole SOAP. Par exemple, d'après le protocole SOAP 1.2, les exceptions utilisées en programmation orientée objet, devraient être traduites en éléments XML appelés *fautes SOAP*. Mais cette fonction n'est pas automatisée et dépend de la façon dont est implanté le service web.

Face à cette difficulté, nous effectuons une analyse préliminaire de la robustesse des services web dans le but de déterminer quels aléas restent vraiment utiles. Nous montrons ainsi que peu d'entre eux sont pertinents, car la plupart sont bloqués par les processeurs SOAP. Ils ne sont donc pas traités par le service web lui-même. Nous analysons aussi les réponses obtenues en présence d'aléas. Nous en déduisons comment effectuer la phase de test. La génération des cas de test peut se résumer en deux phases. Premièrement, nous

*Ce travail est en parti soutenu par l'Agence Nationale de Recherche française à travers le projet WebMov Project http://webmov.lri.fr.

complétons la spécification afin d'y ajouter le comportement non spécifié. Puis les cas de test sont construits en injectant des valeurs prédéfinies. Ces cas de test sont ensuite exécutés grâce à une plateforme de test basée sur un service web, qui a été implantée dans un outil académique.

Ce document est structuré comme suit: la section 2 présente un aperçu sur le paradigme du service web. Nous présentons quelques travaux sur le test des services web ainsi que les motivations de notre approche. La section 3 analyse la robustesse des services web à travers la couche SOAP. La section 4 décrit la méthode de test: nous détaillons la génération des cas de test et le framework d'exécution. Enfin, la section 5 présente quelques perspectives.

2 Un aperçu sur le paradigme du service web

2.1 Les services web

Les services web sont "des applications modulaires, qui se suffisent à elles-mêmes, qui peuvent être publiées, distribuées sur Internet" (Tidwell 2000). Afin de rendre les services web interopérables, l'organisation WS-I a proposé des profils, en particulier le profil WS-I Basic (WS-I 2006). Celui-ci est composé de quatre grands axes: la *description de l'interface du service web* grâce au langage WSDL (Web Services Description Language), la *sérialisation des messages* transmits via le protocole SOAP (Simple Object Access Protocol), l'*indexation des services web* dans des registres UDDI (Universal Description, Discovery Integration) et *la sécurité des services web*, obtenue essentiellement grâce à des protocoles d'authentification et de cryptage.

- la *description du service web* présente les informations nécessaires pour invoquer un service, en définissant les interfaces du service (*endpoints*), les opérations fournies par le service, et les paramètres/réponses de ces opérations. Cette description appelée fichier WSDL (*Web Services Description Language*), montre aussi comment les messages doivent être structurés en définissant les types complexes utilisés par ces derniers,
- la *définition et la construction des messages XML*, sont basées sur le protocole SOAP (*Simple Object Access Protocol*) (SOAP 2003). SOAP est utilisé pour invoquer les opérations (les méthodes) de services web dans un réseau en sérialisant/désérialisant les données,
- la *découverte du service* dans des registres UDDI. Les descriptions de service web sont réunies dans ces registres UDDI (*Universal Description, Discovery Integration* qui peuvent être consultés manuellement ou automatiquement en utilisant des API de programmation pour rechercher dynamiquement un services web spécifique,
- La *sécurité du service*, est obtenue grâce aux protocole HTTPS et au chiffrage XML.

Dans cet article, nous considérons les services web comme des boîtes noires à partir desquelles seuls les messages SOAP sont observables. La définition d'un service web, donnée ci-dessous, décrit les opérations accessibles et les types de paramètre/réponse manipulés par ces opérations. Nous utilisons également la notion de faute SOAP. D'après le protocole SOAP v1.2 (SOAP 2003), une faute SOAP représente un message d'avertissement à l'application cliente indiquant qu'une erreur est survenue. Une faute SOAP est principalement composée d'un code de faute, d'un message, d'une cause et d'éléments XML réunissant les paramètres et détails sur l'erreur déclenchée. Généralement, une faute SOAP est reçue, dans la programmation orientée objet, après le déclenchement d'une exception par le service web. Ces fautes SOAP ne sont pas forcément décrites dans le fichier WSDL.

Définition 2.1 *Un service web WS est un composant qui peut être appelé via un ensemble d'opérations $OP(WS) = \{op_1, ..., op_k\}$, avec op_i défini par $(resp_1, ..., resp_n) = op_i(param_1, ..., param_m)$, où $(param_1, ..., param_m)$ sont les types de paramètres et $(resp_1, ..., resp_n)$ sont les types de réponses.*

Pour une opération op, nous définissons $P(op)$ l'ensemble des tuples de paramètres de op, $P(op) = \{(p_1, ..., p_m) \mid p_i$ est une valeur de type $param_i\}$. De même, $R(op)$ est l'ensemble des tuples de réponses, $R(op) = \{(r_1, ..., r_n) \mid r_j$ est une valeur de type $resp_j\} \cup \{r \mid r$ est de type fauteSOAP $\} \cup \{\epsilon\}$. ϵ représente une réponse vide (ou pas de réponse).

L'opération op correspond à une Relation op : $P(op) \rightarrow R(op)$. Nous notons une invocation de cette opération $r = op(p)$ avec $r \in R(op)$ et $p \in P(op)$.

Chaque paramètre/réponse peut être de type simple (integer, float, String...) ou complexe (arbre, tableau, faute SOAP, objet composé de types simples ou complexes...) et chaque type peut être fini (Integer,...) ou infini (String,...).

La persistance du service web, et donc son état interne, au fil des appels d'opération, est modélisée grâce à un automate symbolique nommé STS (Symbolic Transition System (Frantzen, Tretmans & Willemse 2005)), qui est un formalisme proche des UML state machine. Un automate STS $< L, l_0, Var, var_0, I, S, \rightarrow >$, est composé d'un alphabet de symboles S: les entrées, commençant par "?",

sont fournies au système, tandis que les sorties, commençant par "!", sont reçues et observées depuis le système. Un STS est aussi composé d'un ensemble de variables Var, initialisé par var_0. Chaque transition $(l_i, l_j, s, \varphi, \varrho) \in \rightarrow$ partant de l'état l_i vers l_j étiquetée par le symbole s, peut effectuer une mise à jour des variables avec ϱ et possède une garde φ sur Var qui doit être satisfaite pour tirer la transition.

Nous notons de plus $(x_1, ..., x_n) \models \varphi$, la satisfaction de la garde φ suivant les valeurs $(x_1, ..., x_n)$.

Un exemple de spécification est donnée en figure 1. Celle-ci décrit un service web avec lequel il est possible de se logger de chercher des livres et de les acheter. La base de données associée est donnée en figure 3.

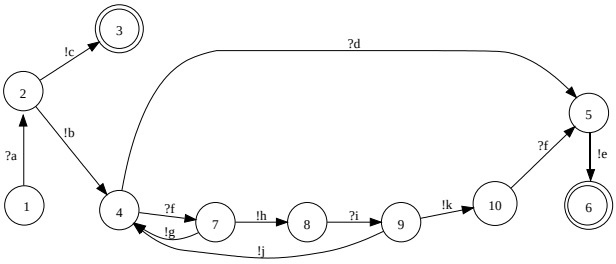

Figure 1: Spécification d'un service web persistant

?a	login<String s>
!c	login_return<String s2> [S2=="invalid" && s ≠ BD.account.name]
!b	login_return<String s2> [S2=="valid" && s == BD.account.name]
?d	disconnect()
!e	disconnect_return<String r> [r=="disconnected"]
?f	list_book<String title>
!g	list_book_return<String r> [r==soapFault("invalid book")&& title ≠ BD.book.title]
!h	list_book_return<String isdn> [isdn==BD.book.isdn(title) && title == BD.book.title]
?i	buy<String i> i:=isdn
!j	buy_return<boolean b> [b==false && fund ≤ book.isdn(isdn).price
!k	buy_return<boolean b> [b==true && BD.account.name(s).fund > book.isdn(isdn).price]

Figure 2: Table des symboles de la spécification

account		
name	fund	
"name"	10	
book		
title	isdn	price
"fundation"	"0553293354"	20

Figure 3: Base de données

2.2 Travaux sur le test de services web

Plusieurs travaux sur le test de services ont été proposés. Certains se sont orientés sur les tests de conformité, de robustesse et d'interopérabilité des services web en boite noire. Dans (Offutt & Xu 2004, Looker, N., Munro, M., Xu & J. 2004), la robustesse des services web est testée en appliquant des mutations sur les messages de requête, puis en analysant les réponses obtenues. Si le service ne répond pas, il est considéré comme non robuste. Dans (Frantzen, Tretmans & de Vries 2006), la spécification décrit des invocations successives de différentes opérations d'un même service. Cette spécification est traduite en un modèle LTS à partir duquel les cas de test sont générés suivant la relation d'implantation *ioco* (Tretmans 1996). Dans (Bai, Dong, Tsai & Chen 2005), les services web sont automatiquement testés en utilisant la description WDSL. Les cas de test sont construits pour analyser des types de données reçues et de tester les dépendances d'opérations (analyse sur des types de paramètres donnés et les types de réponses reçues). Dans (Bertolino, Frantzen, Polini & Tretmans 2006), les auteurs testent l'interopérabilité des services web en proposant d'augmenter la description WSDL via un diagramme PSM (UML2.0 Protocol State Machine), dont le but est de modéliser les interactions possibles entre un client et un service web. Les cas de test sont générés à partir des PSM. Un framework de test, appelé l'"Audition framework", est proposé dans (Bertolino & Polini 2005). Les auteurs de (Martin & Xie 2006) proposent une méthode automatique de test de robustesse des services web. À partir de la description WSDL, la méthode utilise le framework "Axis 2" pour générer une classe cliente composée d'une liste de méthodes permettant d'invoquer toutes les opérations. Ensuite, les cas de test sont générés avec l'outil *Jcrasher* à partir de la classe précédente. Enfin, l'outil *Junit* est utilisé pour exécuter les cas de test. Dans (Vieira, Laranjeiro & Madeira 2007), la robustesse est testée en appliquant des injections de fautes au niveau des paramètres. Le service est robuste s'il retourne uniquement des réponses satisfaisant la description WSDL. La méthode présentée dans (Gorbenko & al. 2008) discute sur une technique d'évaluation de la performance des services web en prenant en compte la gigue d'un réseau. L'article (Bartolini, Bertolino, Marchetti & Polini 2009) présente l'outil WS-TAXI, obtenu en intégrant "SOAPIU" qui est un outil de test manuel permettant d'effectuer des tests sur des opérations de services Web, et "TAXI" qui permet de définir des instances XML à partir d'un fichier XSD. L'outil WS-TAXI permet ainsi de générer des cas de test pour tester automatiquement des opérations en traitant toute la structure du service web comme un fichier XSD.

Dans (Salva & Rabhi 2009), nous avons proposé une

étude sur la robustesse des services web. Celle-ci consiste à évaluer automatiquement la robustesse d'un service web (non persistant) par rapport aux opérations déclarées dans la description WSDL, en examinant les réponses reçues lorsque ces opérations sont invoquées avec des aléas. Un outil a été développé et expérimenté sur des services web réels.

Par rapport aux articles précédents dont le thème est la robustesse, nous considérons un service web persistant dont l'état interne est modélisé par un STS. A la différence des travaux traitant de la robustesse de systèmes "classiques", nous considérons les spécificités des services web et notamment la couche SOAP et la gestion des exceptions (fautes SOAP). Par rapport aux travaux de (Offutt & Xu 2004, Martin & Xie 2006, Vieira et al. 2007) sur le test de robustesse des services web, nous analysons et déterminons les aléas qui peuvent être réellement utilisés. Nous séparons également le comportement du service web et du processeur SOAP, ce qui augmente la détection des erreurs de robustesse. Notamment, nous considérons, comme robuste, un service web retournant des fautes SOAP construites par lui même et non générées par le processeur SOAP. Par rapport à (Martin & Xie 2006), notre architecture de test effectue des appels de service web directement, ce qui permet d'analyser les réponses SOAP, et en particulier les fautes SOAP.

3 Étude de la robustesse de services web

Tout comme dans (*IEEE Standard Glossary of Software Engineering Terminology* 1999), nous considérons qu'un service web est robuste "s'il est capable de fonctionner correctement en présence de fautes ou d'un environnement stressant". Ceci implique qu'un service web robuste doit agir conformément à sa spécification en présence d'aléas. Le challenge apporté par les services web concerne principalement la couche SOAP. En effet, celle-ci réduit l'observation et le contrôle que l'on peut effectuer, notamment en présence d'erreurs. Et c'est ce point qui va être étudié par la suite.

De même que dans le profil WS-I basic, nous considérons qu'un *récepteur* est une partie logicielle qui consomme un message (processeur SOAP + service web). Le processeur SOAP représente souvent une partie d'un framework, comme *Apache Axis* ou *Sun Metro JAXWS*. Celui-ci gère les appels des services web en serialisant/deserialisant les messages. Nous avons observé que les processeurs SOAP affectent grandement le contrôle et l'observation des appels d'opérations.

3.1 Robustesse d'une opération et observabilité

Nous avons analysé dans (Salva & Rabhi 2009) la robustesse des opérations de service web non persistants et avons déduit quels aléas pouvaient être utilisés.

Nous avons analysé les aléas suivant: **Remplacement de types de paramètres, Inversion de types de paramètres, Ajout/injection de types de paramètres, Suppression de types de paramètres** et **Utilisation de valeurs inhabituelles**. Ce dernier reste le seul aléa utilisable car c'est le seul qui n'est pas bloqué par les processeurs SOAP. L'aléa **Utilisation de valeurs inhabituelles**, bien connu dans le test du logiciel (Kropp, Koopman & Siewiorek 1998), peut être employé pour appeler une opération $(resp_1, ..., resp_n) = op(param_1, ..., param_m)$ avec des valeurs $(p_1, ..., p_m) \in P(op)$, tel que chaque paramètre p_i a le type $param_i$. Mais ces valeurs prédéfinies sont inhabituelles. Par exemple null, "", "$", "*" sont des valeurs inhabituelles du type "String". Celles-ci sont acceptées par les processeurs SOAP car elles satisfont la description WSDL du service web.

Nous avons aussi analysé les réponses possibles en présence d'aléas afin de séparer les réponses produites par le service web de celles produites par les processeurs SOAP. En effet, lorsqu'une erreur se produit, certains processeurs SOAP (comme celui d'Axis2 par exemple) génèrent eux-mêmes une faute SOAP à la place du service web. Un tel comportement complique alors le test car on peut estimer que l'opération réagit correctement et est robuste alors que c'est une autre partie logicielle qui répond. De plus, certains processeurs SOAP génèrent des fautes SOAP, dans certains cas, alors que d'autres ne le font pas.

Le profil WS-I basic permet de différencier les fautes SOAP générées par un processeur SOAP des fautes SOAP construites par un service web. En effet, lorsqu'une exception est levée, le service web devrait générer lui-même une faute SOAP. Et dans ce cas, la faute SOAP sera composée de la cause "RemoteException". Autrement, les fautes SOAP sont générées par les processeurs SOAP.

Nous avons déduit de cette analyse qu'une opération est robuste si elle retourne une réponse "classique" ou une faute SOAP composée de la cause "RemoteException". Ceci est formalisé dans la définition suivante:

Définition 3.1 *Soit WS un service web WS. Une opération $(resp_1, ..., resp_n) = op(param_1, ..., param_m) \in OP(WS)$, est robuste en présence de l'aléa "Utilisation de valeurs inhabituelles", ssi $\forall v \in P(op), r = op(v)$ avec:*

- *$r = (r_1, ..., r_n)$ tel que $r_i = resp_i$,*
- *ou r est une faute SOAP composée de la cause "RemoteException".*

3.2 Analyse de la robustesse des services web persistants

Nous avons étudié précédemment la robustesse des opérations de service web et avons déduit dans (Salva & Rabhi 2009) que seul l'aléa **Utilisation de valeurs inhabituelles** pouvait être appliqué. Nous étendons ici cette étude aux services web persistants ayant différents états internes au fil des appels. Tout comme précédemment, certains aléas sont inutiles car bloqués par les processeurs SOAP. Nous avons étudié les aléas suivants: **Modification du nom d'une opération, Remplacement/Ajout du nom de l'opération**. Soient donc un service web WS et STS sa spécification:

- **Modification du nom d'une opération**: cet aléa correspond à la modification aléatoire du nom d'une opération $op \in OP(WS)$ tel que le nom modifié op_modif n'est pas celui d'une opération existante ($op\_modif \notin OP(WS)$). En appliquant ce type d'aléa, nous obtenons toujours une faute SOAP composée de la cause "Client". Celle-ci est construite par les processeurs SOAP et signifie que l'appel client est erroné. L'appel est donc bloqué par les processeurs SOAP et n'est pas propagé vers le service web. Le test ne pouvant pas être effectué, nous considérons que cet aléa est inutile,
- **Remplacement du nom /Ajout d'une opération**: soit l un état de la spécification STS avec les transitions sortantes $(l, l_1, "op_1(p_{11}, ..., p_{m1})", \varphi_1, \varrho_1)$, ..., $(l, l_n, "op_n(p_{1n}, ..., p_{mn})", \varphi_n, \varrho_n)$ modélisant des appels d'opérations. S'il existe une opération $(resp_1, ..., resp_n) = op(param_1, ..., param_m) \in OP(WS)$ qui n'est pas appelée depuis l ($op \notin \{op_1, ..., op_n\}$), cet aléa a pour but de forcer l'appel de l'opération op à la place d'une autre. En appelant op, on ne peut bien sur qu'obtenir une réponse d'op. Si dans notre exemple de la figure 1, nous remplaçons l'opération "login" par "buy", nous n'obtiendrons pas une réponse de "login" mais de "buy". On ne peut donc pas juste remplacer/ajouter un nom. Ainsi, si l'opération op est robuste, selon la définition 3.1, la réponse attendue est soit une réponse "classique" $(r_1, ..., r_n)$ avec r_i de type $resp_i$, soit une faute SOAP dont la cause est "RemoteException".

  Cet aléa implique donc la modification de la spécification pour la compléter sur les réponses attendues. Cette modification est détaillée lors de la génération des cas de test.

Il existe bien sûr d'autres aléas comme le remplacement du nom du service appelé ou la modification aléatoire des messages SOAP. Ces aléas sont généralement utilisés pour le test de services web composés afin d'observer le comportement global des partenaires inclus dans la composition. Ceux-ci ne sont pas des plus intéressants pour tester un service web unique. A noter que le profil WS-I basic ne permet pas la surcharge d'opération donc cet aléa n'est pas étudié.

Par conséquent, les seuls aléas qui montrent de la pertinence et que nous utiliserons dans la suite sont l'Utilisation de valeurs inhabituelles et le Remplacement du nom /Ajout d'une opération.

4 Méthode de test de robustesse de services web persistants

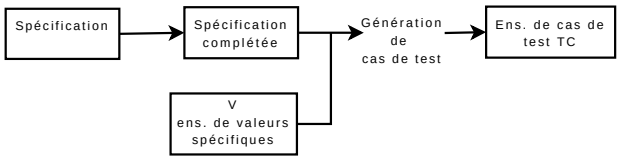

Figure 4: Génération des cas de test

L'analyse précédente sur la robustesse des opérations nous montre que les aléas les plus pertinents sont le "Remplacement du nom /Ajout d'une opération" et l' "Utilisation de valeurs inhabituelles". Ces deux aléas sont donc utilisés au cour de la génération des cas de test. La méthode est illustrée en figure 4. La spécification est premièrement complétée sur les opérations qui peuvent être appelées en utilisant l'aléa "Remplacement du nom /Ajout d'une opération". Elle est aussi complétée afin de modéliser les comportements incorrects sur la quiescence des états (états bloqués après un timeout) et sur l'ensemble des réponses incorrectes.

Les cas de test sont ensuite générés en utilisant l'aléa "Utilisation de valeurs inhabituelles". Un ensemble de valeurs prédéfinies, noté V, est donc employé. Celui-ci est composé pour chaque type de données, d'une liste XML de valeurs spécifiques qui ont été choisies afin d'effectuer des appels avec des valeurs inhabituelles. Nous utilisons des valeurs connues dans le test de logiciel, qui sont supposées avoir un taux de déclenchement d'erreurs élevé (Kropp et al. 1998).

Nous notons $V(t)$ l'ensemble des valeurs de type t qui sont utilisées pour invoquer une opération de service web. Par exemple, la figure 5 décrit quelques valeurs utilisées pour le type "String" et pour le type "tabular de "simple-type". Pour un "tabular" de "String" nous utiliserons donc les valeurs suivantes: "tabular" vide, "tabular" avec éléments vides et plusieurs "tabular" construite avec $V(String)$.

Afin de tester la robustesse des opérations, nous

```
<type id="String">
        <val value=null />
        <val value="" />
        <val value=" " />
        <val value="$" />
        <val value="*" />
        <val value="&" />
      <val value="hello" />
        <val value=RANDOM" />
        <!-- a random String-->
        <val value=RANDOM(8096)" />
</type>
```

(a) V(String)

```
<type id="tabular">
        <val value=null />
        <!-- an empty tabular-->
        <val value= null null />
        <!--tabular
        composed of two empty elts-->
        <val value= simple-type />
</type>
```

(b) V(tabular)

Figure 5

définissons aussi l'hypothèse suivante sur les services web. Celle-ci nous permet de toujours différencier une opération robuste d'une autre qui ne l'est pas par l'observation d'une réponse.

Hypothèse d'observabilité d'une opération du service web: nous supposons que chaque opération du service web, décrite dans des fichiers WSDL, renvoie des réponses non nulles.

Par suite, nous développons en détails la complétion de la spécification, la génération des cas de test et l'élaboration du verdict final.

4.1 Complétion de la spécification

Comme énoncé précédemment, nous complétons la spécification pour y injecter l'aléa "Remplacement du nom /Ajout d'une opération" et pour décrire tous les comportements corrects et incorrects du service web. Soit donc le service web WS et sa spécification $STS =< L, l_0, Var, var_0, I, S, \rightarrow >$. STS est complétée grâce aux étapes suivantes:

1. **remplacement des conditions sur base de données** par des valeurs réelles,
2. **ajout du verdict "pass" dans les états finaux**, ce qui illustre que pour atteindre cet état final, un comportement correct a été exécuté,
3. **complétion sur les appels d'opérations**: $\forall(l, l', op(p_1, ..., p_m), \varphi, \varrho) \in \rightarrow$ nous ajoutons $\forall op_i \neq op \in OP(WS)$ $(l, l_i, op_i(p_{i1}, ..., p_{im}), \emptyset, \emptyset)$, $(l_i, l, op_i\_return(r_1), \varphi_i, \emptyset)$, avec $\varphi_i = [r_1 \neq ((c_1, soapfault), c_1 \neq RemoteException)]$. Ceci correspond à l'aléa "Remplacement du nom /Ajout d'une opération". Si, à partir d'un état, une opération non spécifiée est appelée et que la réponse obtenue est soit composée d'une valeur "classique" soit est une faute SOAP dont la cause est "RemoteException" alors le service web a accepté l'appel et a répondu de façon robuste (voir section 3.2). Il revient ensuite à l'état précédent l'appel,
4. **complétion sur la réception de réponses incorrectes**: $\forall l \in L$ tel que l a des transitions sortantes $(l, l_1, "op\_return(r_1)", \varphi_1, \varrho_1)$, ..., $(l, l_n, "op\_return(r_n) \quad ", \varphi_n, \varrho_n)$, nous ajoutons: (1) $(l, fail, \delta, \emptyset, \emptyset)$, (2) $(l, fail, "op\_return(r)", \neg(\varphi_1||...||\varphi_n), \emptyset)$. Cette complétion modélise le manque de robustesse suivant les réponses observées. (1) Si l'état est quiescent (bloquant), le service web est figé et l'opération appelée n'est pas robuste. (2) Si l'opération appelée ne retourne pas une réponse spécifiée, alors le comportement du service web est incorrect en présence d'aléas et donc l'opération n'est pas robuste.

En partant de la spécification de la figure 1, la spécification complétée est obtenue en figure 6. Les transitions modélisées par des pointillés représentent l'appel d'opérations non spécifiées. Celles illustrées avec des traits hachurés correspondent aux réponses modélisant un comportement erroné.

4.2 Génération des cas de test

Avant de décrire leur génération, nous définissons un cas de test par:

Définition 4.1 *Soit WS un service web modélisé par $STS =< L, l_0, Var, var_0, I, S, \rightarrow >$. Un cas de test T est un arbre où chaque nœud terminal est étiqueté par un verdict dans $\{pass, fail\}$. Une branche est étiquetée soit par $op(v), \varphi, \varrho$ soit par $op\_return(r), \varphi, \varrho$, soit par δ avec:*

- *$v \in P(op)$, est une liste de valeurs utilisée pour invoquer op,*
- *$r = (c, soap\_fault)$ est une faute SOAP composée de la cause c,*
- *$r = (r_1, ..., r_m)$ est une réponse,*
- *φ et ϱ une condition et une mise à jour sur l'ensemble des variables de Var respectivement,*
- *δ représente la quiescence d'un état, (état bloquant après un timeout).*

Les cas de test sont construits grâce à l'algorithme suivant. Celui-ci génère des traces dans lesquelles

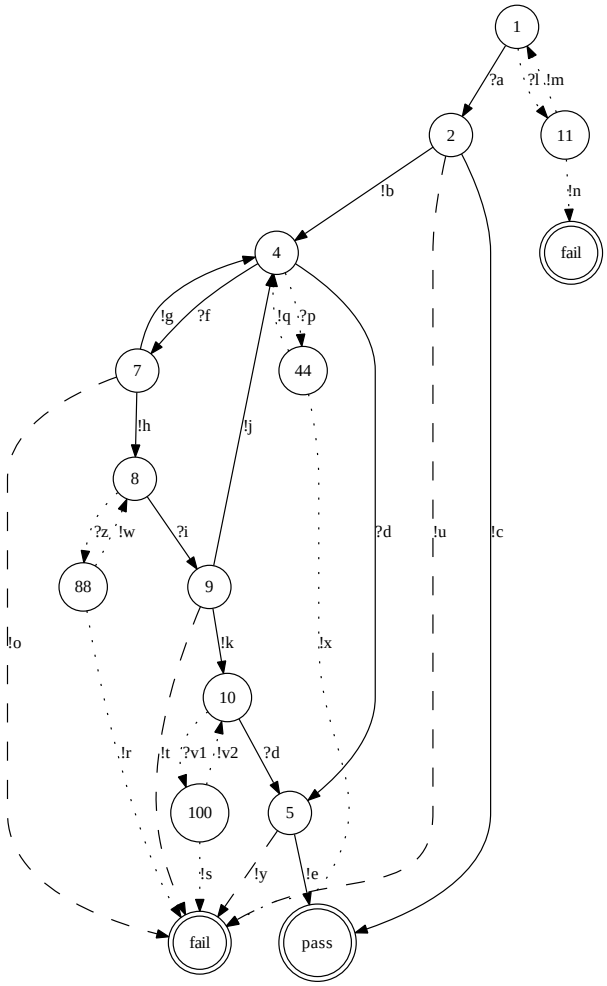

Figure 6: Spécification complétée

!c	login_return<String s2> [S2=="invalid" && s ≠ "name"]
!b	login_return<String s2> [S2=="valid" && s == "name"]
!g	list_book_return<String r> [r==soapFault("invalid book")&& title ≠ "fundation"]
!h	list_book_return<String isdn> [isdn=="0553293354" && title == "fundation"]
!j	buy_return<boolean b> [b==false && fund ≤ 20]
!k	buy_return<boolean b> [b==true && fund > 20]
?l	list_book<String title>|disconnect()|buy<String isdn>
!m	list_book_return(r)|disconnect_return(r)|buy_return(r) r=String|r=Boolean'|r=(c,soapFault) c="RemoteException"
?n	δ|list_book_return(r)|disconnect_return(r)| buy_return(r) r=(c,soapFault) c≠"RemoteException"
?p	login<String s>|buy<String isdn>
!q	login_return(r)|buy_return(r) r=String|r=(c,soapFault) c="RemoteException"
!x	δ|login_return(r)|buy_return(r) r=(c,soapFault) c≠"RemoteException"
?z	list_book<String title>|disconnect()|login<String s>
!w	list_book_return(r)|disconnect_return(r)| login_return(r) r=String|r=(c,soapFault) c="RemoteException"
!r	δ|list_book_return(r)|disconnect_return(r)| login_return(r) r=(c,soapFault) c≠"RemoteException"
?v1	list_book("title")|buy("isdn")|login("name")
!v2	list_book_return(r)|buy_return(r)|login_return(r) r=String|r=Boolean|r=(c,soapFault) c="RemoteException
!s	δ|list_book_return(r)|buy_return(r)|login_return(r) r=(c,soapFault) c≠"RemoteException"
!u	δ|login_return(r) [(S2 ≠ "invalid" || s =="name") && (S2 ≠ "valid" ||s ≠ "name")]
!y	δ|disconnect_return(r) [r≠"disconnected"]
!o	δ|list_book_return(r) [(r≠soapFault("invalid book")||title=="fundation") && (r≠"0553293354" || title ≠ "fundation")]
!t	δ|buy-return(r) [(b ≠ false || fund > 20)&&(b ≠ true || fund ≤ 20)]

Figure 7: Table des symboles de la spécification détaillée

sont injectés des aléas. Pour une transition modélisant un appel d'opération *op*, l'algorithme construit un préambule permettant d'atteindre cette transition (lignes 4-5). Un solveur de contraintes (Niklas Een 2003, Kiezun, Ganesh, Guo, Hooimeijer & Ernst 2009) est utilisé pour obtenir des valeurs permettant sa complète exécution. Nous ajoutons $op(v_1, ..., v_m)$ au préambule pour appeler l'opération *op* avec des valeurs inhabituelles $(v_1, ..., v_n)$ (ligne 9). Puis, nous concaténons toutes les traces exécutables (dont les conditions peuvent être satisfaites) atteignant un état final étiqueté par un verdict (lignes 10-11).

1 **Algorithme :**Génération des cas de test
2 Testcase(STS): TC
3 **pour chaque** *transition* $t = (l_k, l_{k+1}, ?e_k, \varphi_k, \varrho_k)$ *étiquetée par une entrée* $?e_k = op(param_1, ..., param_m)$ **faire**
4 $path\ p = DFS(l_0, l_k)$
5 Resolution(p)
6 $Value(op) = \{(v_1, ..., v_m) \in V(param_1) \times ... \times V(param_m)\}$
7 **pour chaque** $(v_1, ..., v_m) \in Value(op)$ **faire**
8 $TC = TC \cup tc$ avec
9 $tc = p; (l_k, l_{k+1}, op(v_1, ..., v_m), \varphi_k, \varrho_k); t$
10 avec $t = t'_j; postambule; verdict$ tel que $\forall t' = (l_{k+1}, l_j, a_j, \varphi_j, \varrho_j) \in \rightarrow, (v_1, ..., v_n) \models \varphi_j$
11 et $postambule = DFS(l_j, l_t)$ est un chemin entre l_j et un état final $l_t \in L$
12 et *verdict* étiqueté dans l_t
13 Resolution(postambule)
14 **fin**
15 **fin**
16 Resolution(path p):p
17 $p = (l_0, l_1, a_0, \varphi_0, \varrho_0)...(l_{k-1}, l_k, a_{k-1}, \varphi_{k-1}, \varrho_{k-1})$
18 **pour chaque** $(l_i, l_{i+1}, a_i, \varphi_i, \varrho_i)$ *avec* $i > 0$ **faire**
19 $(x_1, ..., x_n) = solver(\varphi_i)$ //résolution de contraintes sur les variables $(X_1, ..., X_n)$ utilisée par φ_i grâce à un solveur
20 $\varrho_{i-1} = \varrho_{i-1} \cup \{X_1 = x_1, ..., X_n = x_n\}$
21 **fin**

Algorithme 1 : Génération des cas de test

Les solveurs de contraintes permettent de rechercher des valeurs satisfaisant les conditions d'un chemin de la spécification et donc satisfaisant son exécution. Nous utilisons les solveurs (Niklas Een 2003) et (Kiezun et al. 2009). Ceux-ci fonctionnent sous forme de serveurs externes peuvent être facilement appelés par l'algorithme de génération de cas de test. Le solveur (Kiezun et al. 2009) traite les types "String", le solveur (Niklas Een 2003) peut notamment traiter les types "booléen" et "Integer". Dans le cas où le type de donnée est complexe (objet, ...) la résolution de contraintes doit être faite manuellement.

Les figures 8 et 9 illustrent des exemples de cas de test obtenus à partir de la spécification complète de la figure 6. Dans le premier, l'opération "login" est appelée avec l'aléa "&". Si la réponse obtenue est "invalid" (seule réponse possible avec l'aléa donné) alors le verdict est "pass", autrement on obtient "fail". Dans le deuxième cas de test, l'opération "disconnect" est appelée à la place de "login". Le service est robuste s'il répond "classiquement" ou avec une faute SOAP composée de la cause "RemoteException" et s'il est toujours possible d'utiliser l'opération "login" pour atteindre un état final "pass" de la spécification (fonctionnement normal après l'utilisation de l'aléa). Autrement, le verdict est "fail" et le service web n'est pas robuste.

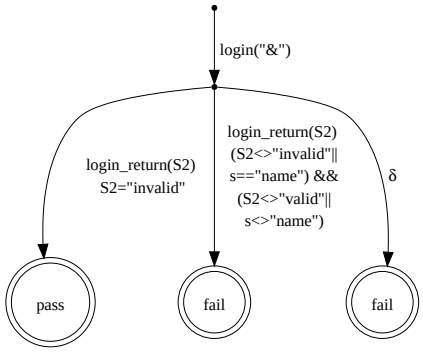

Figure 8: Cas de test: utilisation d'un aléa

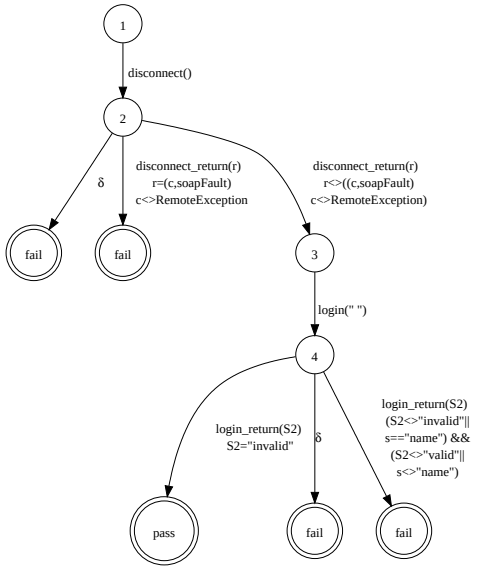

Figure 9: Cas de test: remplacement d'une opération

4.3 Exécution des cas de test

Les cas de test sont générés puis exécutés grâce à la plateforme de test illustrée en figure 10. Celle-ci a été implantée dans un outil académique. Le testeur correspond à un service web qui reçoit l'URL du service

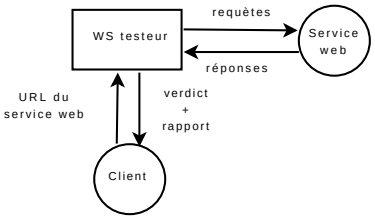

Figure 10: Plateforme de test

web à tester et la spécification STS. Le testeur construit les cas de test comme décrit en section 4.2, puis les exécute successivement sur le service web persistant. Une fois les cas de test exécutés, l'analyse des réponses obtenues est effectuée pour produire un verdict final de test.

Un cas de test étant modélisé par un arbre, le testeur l'exécute en le parcourant et en appelant les opérations correspondantes. Si une branche est complètement exécutée alors le verdict local "pass" ou "fail" est retourné. Sachant qu'une opération non robuste, peut ne pas retourner de réponse, nous avons considéré que chaque noeud d'un cas de test est quiescent si aucune réponse n'est reçue dans les 60s. Dans ce cas, le verdict "fail" est donné (en suivant une branche étiquetée par δ). Soit t un cas de test et $trace(t) \in \{\text{pass} , \text{fail} \}$ le verdict local associé. Le verdict final est donné par:

Définition 4.2 *Soit WS un service web et TC un ensemble de cas de test. Le verdict de test par rapport à TC, noté $Verdict(WS)/TC$ est:*

- *"pass", si pour tout $t \in TC, trace(t) = pass$,*
- *"fail", s'il existe $t \in TC$ tel que $trace(t) = fail$.*

5 Conclusion

Nous avons proposé, dans cet article, une méthode de test de robustesse de services web persistants modélisés par des spécifications symboliques STS. L'une des particularités des services web concerne leur imbrication dans un environnement SOAP, qui réduit et modifie le contrôle et l'observation des messages. Nous améliorons cette observabilité en séparant les types de fautes SOAP observées. Ainsi, notre méthode détecte les fautes SOAP non générées par les services web eux-même mais par les processeurs SOAP, ce qui améliore la détection de problèmes de robustesse. De même, avons analysé le comportement des services web en présence d'aléas, et avons conclu que seuls les aléas "Remplacement du nom /Ajout d'une opération" et l' "Utilisation de valeurs inhabituelles" sont utiles car ce sont les seuls qui sont réellement propagés jusqu'aux services web. Notre méthode de test prend donc en compte ces aléas en complétant le comportement de la spécification et en injectant ces valeurs inhabituelles dans les cas de test.

Cette méthode, comme toutes celles basées sur des types complexes, possède un inconvénient inhérent aux solveurs de contraintes. En effet, la génération des cas de test s'effectue automatiquement grâce à ces solveurs de contraintes qui peuvent manipulés plusieurs types simples. Il n'existe cependant pas de solveurs sur des types complexes. Dans ce cas, la seule solution est de construire à la main ces valeurs. Cependant, nous avons observé qu'une grande majorité des services web existants restent basés sur des types simples.

Plusieurs perspectives futures peuvent être envisagées notamment par rapport à l'ensemble de valeurs V. Celui-ci peut en effet être modifié mais reste statique lors de la construction des tests. Il pourrait être plus intéressant de proposer une analyse dynamique pour construire une liste de valeurs inhabituelles la plus appropriée pour chaque service web. Pour éviter l'explosion des cas de test, les paramètres sur V sont choisis aléatoirement. Une meilleure solution serait de choisir ces paramètres selon la description de l'opération ou bien d'analyser les valeurs relevant le plus d'erreurs pendant les tests et d'y apposer une priorité (coefficient de pondération).

Nous avons également supposé que les messages envoyés et reçus par les services web sont seulement des messages SOAP. Cependant, des services peuvent être reliés à d'autres serveurs, comme des serveurs de base de données. Dans des travaux futurs, nous envisageons donc de considérer le service web comme une boîte grise à partir de laquelle, n'importe quel type de message pourrait être observé.

References

Bai, X., Dong, W., Tsai, W.-T. & Chen, Y. (2005). Wsdl-based automatic test case generation for web services testing, *SOSE '05: Proceedings of the IEEE International Workshop*, IEEE Computer Society, Washington, DC, USA, pp. 215–220.

Bartolini, C., Bertolino, A., Marchetti, E. & Polini, A. (2009). Ws-taxi: A wsdl-based testing tool for web services, *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, IEEE Computer Society, Washington, DC, USA, pp. 326–335.

Bertolino, A., Frantzen, L., Polini, A. & Tretmans, J. (2006). Audition of web services for testing conformance to open specified protocols, *in* R. Reussner, J. Stafford & C. Szyperski (eds), *Architecting Systems with Trustworthy Compo-*

nents, number 3938 in *LNCS*, Springer-Verlag, pp. 1–25.

Bertolino, A. & Polini, A. (2005). The audition framework for testing web services interoperability., *EUROMICRO-SEAA*, pp. 134–142.

Frantzen, L., Tretmans, J. & de Vries, R. (2006). Towards model-based testing of web services, *in* A. Bertolino & A. Polini (eds), *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, Palermo, Sicily, ITALY, pp. 67–82.

Frantzen, L., Tretmans, J. & Willemse, T. (2005). Test Generation Based on Symbolic Specifications, *in* J. Grabowski & B. Nielsen (eds), *Formal Approaches to Software Testing – FATES 2004*, number 3395 in *Lecture Notes in Computer Science*, Springer, pp. 1–15.

Gorbenko, A. & al. (2008). The threat of uncertainty in service-oriented architecture, *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems. ACM.*

IEEE Standard Glossary of Software Engineering Terminology (1999). *IEEE Standards Software Engineering 610.12-1990. Customer and terminology standards*, Vol. 1, IEEE Press.

Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P. & Ernst, M. D. (2009). Hampi: a solver for string constraints, *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, New York, NY, USA, pp. 105–116.

Kropp, N. P., Koopman, P. J. & Siewiorek, D. P. (1998). Automated robustness testing of off-the-shelf software components, *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, Washington, DC, USA, p. 230.

Looker, N., Munro, M., Xu & J. (2004). Ws-fit: A tool for dependability analysis of web services, *in Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC'04). IEEE Computer Society Press, Vol. 02.*

Martin, E. & Xie, T. (2006). Automated test generation for access control policies, *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006).*

Niklas Een, N. S. (2003). Minisat. http://minisat.se.

Offutt, J. & Xu, W. (2004). Generating test cases for web services using data perturbation, *in* S. E. Notes (ed.), *ACMSIGSOFT*, Vol. 29(5), pp. 1–10.

Salva, S. & Rabhi, I. (2009). Automatic web service robustness testing from wsdl descriptions, *12th European Workshop on Dependable Computing.*

SOAP (2003). Simple object access protocol v1.2 (soap), World Wide Web Consortium.

Tidwell, D. (2000). Web services, the web's next revolution, *IBM developerWorks.*

Tretmans, J. (1996). Test generation with input, outputs, and repetitive quiescence, *Software - Concepts and Tools* **17**: 103–120.

Vieira, M., Laranjeiro, N. & Madeira, H. (2007). Assessing robustness of web-services infrastructures, *In Proc. of the Int. Conf. On Dependable Systems and Networks (DSN'2007).*

WS-I (2006). Ws-i basic profile, WS-I organization.