Robustesse des Services Web Composés

Issam Rabhi
Laboratoire Informatique d'Avignon
Université d'avignon, France.
issam.rabhi@univ-avignon.fr

Patrice Laurençot
LIMOS, UMR CNRS 6158
Université de Blaise Pascal, France.
laurenco@isima.fr

Résumé

Nous proposons dans cet article une méthode de test de robustesse de services Web composés. Nous prenons en entrée une spécification symbolique modélisant le comportement du service Web composé. Nous transformons cette spécification en arbre d'exécution symbolique caractérisant les différents chemins d'exécution suivis durant l'exécution de la composition. Ensuite, nous décomposons cet arbre d'exécution symbolique, en un ensemble de chemins par opération. L'avantage de cette décomposition est d'éclater, de façon automatique, cet arbre en objectif de test par opération. Nous générons les cas de test en complétant les objectifs de test pour tester leur conformité sur l'implantation sous test.
Mots clés : Test de Robustesse, Service Web, Objectif de Test, Systèmes Symboliques.

Abstract

We propose in this paper, a robustness testing method of composite Web services. Our method aims to test operations robustness which are provided by the involved Web services in the composition. Symbolic specification, used as a composition model is first translated into a symbolic execution tree which characterizes the execution paths followed during the symbolic execution. Then, from this symbolic execution tree a set of sub-trees per operation are generated. These ones allow to automatically derive useful test purpose by operation. We prepare a set of test cases for each operation, by completing the test purposes to test their conformance againt the implementation under test.
Keywords : Robustness Testing, Web Service, Test purpose, Symbolic System.

1 Introduction

Actuellement, sous le vocable de SOA, se développe un style d'architecture orienté service permettant de construire des systèmes informatiques évolutifs et adaptables par recours à des composants réutilisables appelés services. La composition de services est un des enjeux principaux du SOA. On dira qu'un service Web est composé lorsque son invocation engendre des interactions avec d'autres services Web. Les services Web composés sont souvent utilisés dans des applications vastes et complexes. Afin de produire une application fiable, les entreprises qui utilisent les services Web composés doivent intègrer un ensemble d'activités de tests, comme le test de robustesse. En effet, les services Web composés sont distribués par nature et peuvent être utilisés par des applications clientes. Par conséquent, ils doivent pouvoir se comporter correctement en cas de réception d'événements non prévus. En d'autres termes, ils doivent être robustes. Nous proposons dans cet article une méthode de test de robustesse des opérations fournies par les services Web impliqués dans la composition. Nous décrivons cette composition avec un STS (Système de Transition Symbolique). Tout d'abord, notre méthode transforme le STS en arbre d'exécution symbolique et détermine pour chaque opération l'ensemble des chemins symboliques permettant d'atteindre la réponse retournée par cette opération. Cette transformation nous permet d'avoir un objectif de test par opération et de différencier les opérations décrites dans la spécification globale. Ensuite, la méthode teste la robustesse de chaque opération, c'est-à-dire sa capacité à résister à différentes valeurs inattendues en utilisant un ensemble prédéfini de valeurs [7]. Nous considérons une opération robuste, si les observations faites sur l'exécution

du test sont conformes aux chemins définis dans la spécification. Ce document est structuré comme suit : dans la section 2, nous présentons un aperçu sur les services Web composés. Nous mentionnons quelques travaux sur les tests de services Web ainsi que les motivations de notre approche. La section 3 présente la méthodologie de test de robustesse de service Web composés. Enfin, la section 4 présente les perspectives et les conclusions.

2 Un aperçu sur le Paradigme du Service Web Composé

2.1 Les Services Web Composés

Le développement d'applications à base de services, dont le but est d'assurer une fonctionnalité plus importante, s'appuie sur la composition de ces derniers. Il existe deux stratégies pour composer ces services Web : la chorégraphie et l'orchestration. La chorégraphie définit la collaboration et l'échange de message point à point entre plusieurs partenaires et plusieurs services Web ainsi elle propose une vision globale des interactions. L'orchestration, elle, définit l'enchaînement des services selon un scénario prédéfini, elle propose une vision centralisée décrivant la manière par laquelle les services peuvent agir entre eux. Plusieurs langages ont été proposés pour composer les services tels que le BPEL. Ce dernier favorise l'imbrication des activités et peut être aussi utilisé comme spécification.

D'autres modèles permettant de spécifier des services composés ont été proposés dont les IOLTS (systèmes de transitions à entrée/sortie) [8] et les STS [4]. Dans ce document, nous utiliserons le STS car, à la différence de BPEL, ce modèle permet de présenter la composition de manière aplat.

Un automate STS $< L, l_0, Var, var_0, I, S, \rightarrow$ est constitué d'un alphabet de symboles S dont les entrées, commençant par "?", sont fournies au système, tandis que les sorties, commençant par "!", sont envoyés par le système. Un STS est aussi composé d'un ensemble de variables Var initialisé par var_0, qui est mis à jour en tirant une transition inclus dans I. Chaque transition $(l_i, l_j, s, \varphi, \varrho) \in \rightarrow$ partant de l'état l_i vers l_j étiqueté par le symbole s.

Un chemin symbolique ch est une suite finie de symboles/états qui correspond à un parcours dans l'automate avec $ch = l_0\ s_0\ l_1\ ...\ l_{n-1}\ s_m\ l_n$. Nous définissons : $ACH(op_i)$ comme l'arbre de chemins symboliques pour l'opération op_i, ce qui correspond à l'ensemble des chemins d'un automate dont la dernière transition correspond à $opRep(op_i)$ (la réponse de op_i).

2.2 Travaux sur le test de services Web

Plusieurs travaux sur le test de la robustesse de services Web non composés ont été proposés. Dans [9], 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. Dans [7], nous avons proposé des études sur la robustesse, afin d'évaluer automatiquement la robustesse d'un service Web par rapport aux opérations déclarées dans la description WSDL, en examinant les réponses reçues lorsque ces opérations sont invoquées avec des valeurs aléatoires. Nous séparons également le comportement du service Web et du processeur SOAP, ce qui augmente la détection des erreurs de robustesse. L'approche présentée dans [2] propose une génération de tests avec des données spécifiées par des contraintes telles que les paramètres de qualité de service(QoS).

D'autres travaux sur le test de services Web composés ont été proposés. Les auteurs [5], proposent une méthode de test de robustesse de services Web composés en BPEL qui prend en compte la gestion des fautes (FaultHandler). Les auteurs [10], proposent une approche de

test unitaire pour BPEL en se basant sur Junit. Cette approche génère des chemins de test qui doivent être affinés manuellement par l'ajout des données de test. Dans [6], les auteurs présentent une méthode de test automatique de services composés en BPEL. En transformant la spécification en langage IF, ceci permet de modéliser les contraintes temporelles. Dans [1], les auteurs proposent une méthode de test automatique pour une orchestration en BPEL. La méthode transforme la spécification BPEL en STS afin de générer des cas de test symboliques. Ensuite et à l'aide d'un algorithme de test en ligne, un test de conformité sera réalisé entre la spécification de l'orchestration et l'implantation.

À la différence de [1], nous nous basons sur le test de robustesse et nous séparons le comportement des opérations fournies par les services Web impliqués dans la composition. Nous considérons une nouvelle définition de la robustesse : une opération est considérée robuste si les observations faites sur l'exécution du test sur son implémentation sous test (IUT) sont conformes aux chemins définis dans son arbre de chemin symbolique. En effet, une opération peut être robuste en l'invoquant directement, mais ne pas l'être forcément si elle est invoquée à la suite de plusieurs invocations de différents partenaires. Ainsi notre méthode permet d'affirmer la robustesse d'un service Web par rapport à son environnement.

3 Approche de Test

3.1 Robustesse

Tout comme dans IEEE Standard Glossary of Software Engineering Terminology (1999), nous considérons la robustesse comme étant " la capacité d'un système à fonctionner correctement en présence d'entrées invalides ou d'environnement stressant". Ceci implique qu'un service Web est robuste si ses opérations agissent conformément à sa spécification (WSDL) en présence d'entrées invalides ou aléatoires (aléas).

La relation ioco [8] est une relation de conformité entre implémentations et spécifications qui détermine l'ensemble des implémentations conformes à une spécification donnée. Nous utilisons la formulation ioco (que nous détaillons plus loin) pour définir la robustesse des opérations en analysant des types de données reçues avec le test de la conformité entre l'IUT et ACH. Pour cela, nous définissons trace une trace d'exécution représentant les observations possibles par un testeur sur une IUT, ou encore une séquence finie d'événements observés lors des exécutions des cas de test. Soit :

1. $trace(IUT(op_i))$ est l'ensemble des traces d'exécution de l'IUT pour une opération op_i,
2. $trace(T(ACH(op_i)))$ est l'ensemble des traces d'une suite de test T, dont la suite de test T est l'ensemble des cas de tests possibles associés à $ACH(op_i)$.

3.2 Méthodologie de Test

Notre méthodologie vise à tester automatiquement la robustesse des opérations fournies par les services Web impliqués dans la composition. Elle prend en entrée une spécification STS modélisant le comportement du service Web composé. Tout d'abord, nous transformons la spécification en arbre d'exécution symbolique, caractérisant le comportement des services Web durant l'exécution symbolique de la composition. L'exécution symbolique est une technique utile dans l'analyse de systèmes incluant la génération de cas de test. Ensuite, à partir de cet arbre, nous construisons un sous-arbre de chemins par opération et cela pour toutes les opérations impliquées dans la composition. En outre, un des avantages de cette méthode est qu'elle est automatique puisque pour chaque opération, un sous-arbre de chemins sera préparé.

Ensuite et pour chaque sous-arbre, un ensemble de cas de tests sera généré par le coordinateur sous forme de messages SOAP contenant des valeurs spécifiques. Ces cas de test permettent alors de tester la robustesse de l'opération ciblée. Notre méthodologie de test passe par quatre principales décrites dans la Figure 1.

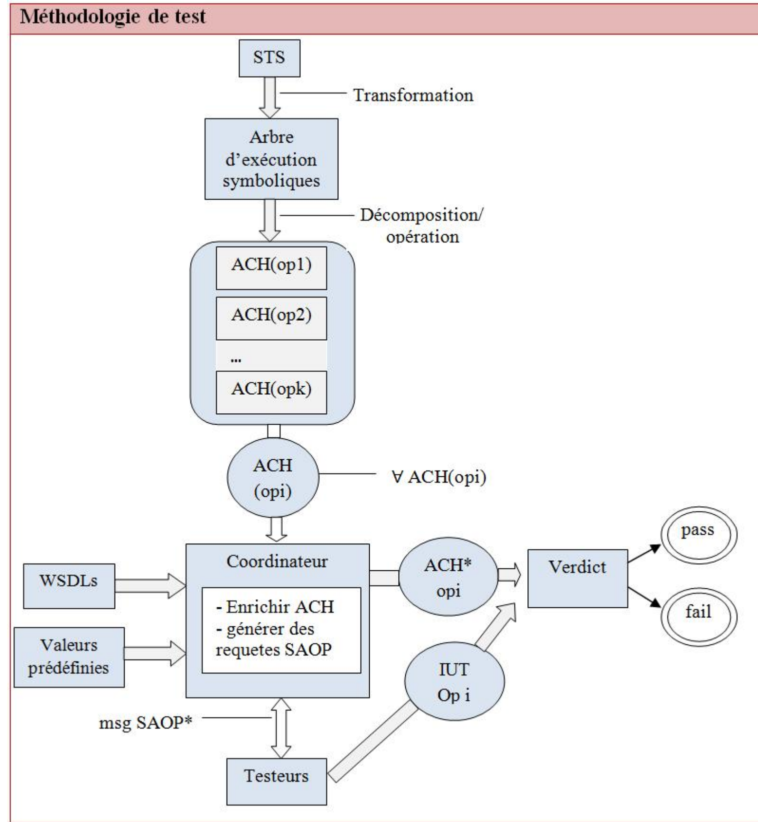

Figure 1: Méthodologie de Test

3.2.1 Transformation de la Spécification en Arbre d'Exécution

La première étape consiste à extraire les comportements représentatifs de notre système de transitions symboliques STS. Pour cela, et à partir de la spécification S, nous générons l'arbre d'exécution symbolique Å [3], dont chaque branche de l'arbre correspond à une exécution possible de notre spécification. Un comportement de la spécification correspond à un chemin. La génération de l'arbre d'exécution symbolique Å est détaillée dans [3].

3.2.2 Décomposition de l'Arbre Symbolique

Tester une opération précise consiste à tester le sous-arbre de l'arbre d'exécution symbolique décrivant tous les chemins et comportements possibles permettant d'atteindre l'opération ciblée. Cette deuxième étape consiste donc à partitionner l'arbre d'exécution symbolique Å en un ensemble de chemins par opération, noté $ACH(op_i)$. Afin de construire $ACH(op_i)$ depuis l'arbre

Å, toutes les transitions accessibles, depuis chaque état e de l'arbre Å, doivent être parcourues pour trouver la transition finale modélisant la réponse de l'opération notée $opRep(op_i)$. Soit

```
Main
    Pour chaque op_i ∈ WS faire //WS : service Web composé
    ACH(op_i).add(l_0)
    rechercherCheminsOP(ACH(op_i), l_0, opRep(op_i), Å)
    fin pour
Fonction rechercherCheminsOp(ACH : in/out, courant : in, transitionF : in, Å : in)
début
    pour chaque etat_i ∈ successeur(courant) faire
        si (courant.transition = transitionF) alors
            ACH.add(etat_i) //add ajoute l'état et la transition
            exit()
        si non
            rechercherCheminsOP(ACH.add(etat_i), etat_i, transitionF, Å)
        fin si
    fin pour
fin
```

Algorithm 1: RECHERCHERCHEMINS

l'algorithme 1 détaillant la construction des arbres symboliques $ACH(op_i)$. L'algorithme est appliqué sur toutes les opérations impliquées dans la composition. Nous obtenons ainsi un arbre symbolique qui peut être transformé en un objectif de test par opération, ce qui permet au coordinateur de différencier les comportements des opérations décrites dans la spécification globale. Puisque pour chaque opération, un sous-arbre de chemins sera préparé. Ensuite et pour chaque sous-arbre, un ensemble de cas de tests (contenant des valeurs spécifiques) sera généré. Ainsi, ces cas de test permettent de tester automatiquement la robustesse de l'opération ciblée.

3.2.3 Génération des séquences de test

Pour un objectif de test donné nous avons besoin de compléter sa spécification, en entrée et en sortie, afin qu'elle puisse être utilisable dans le test. La complétude garantit qu'un blocage de l'interaction entre le testeur et l'IUT induit forcément un verdict. Le but de cette étape est de compléter les arbres de chemins symboliques $ACH(op_i)$ et de préparer les cas de tests. Par le biais de la phase précédente, nous obtenons une liste de requête/réponse permettant d'aller de l'état initial de la spécification vers l'état final recherché. Il nous faut ensuite les compléter avec les différents paramètres. En se basant sur la liste de WSDLs des services Web, le coordinateur enrichit $ACH(op_i)$ en ajoutant les types de paramètres de l'ensemble des opérations afin qu'elle puisse être utilisable dans le test. Ainsi un arbre complété $ACH*(op_i)$ est associé pour chaque $ACH(op_i)$. Enfin, les cas de test sont construits grâce aux solveurs de contraintes. Un cas de test TC est un arbre dont chaque branche décrit une séquence d'interactions entre le testeur et l'IUT, et chaque état feuille à été marqué par $pass$ ou $fail$.

3.2.4 Exécution de Test et Verdict

L'exécution d'un test consiste à soumettre les messages SOAP contenant les données de test à l'IUT et d'observer les réactions de ce dernier. Ensuite, après avoir recomposé la trace de l'exécution de l'implémentation sous test($trace(IUT(op_i))$), le coordinateur se charge de la comparer vis à vis de la $trace(T(ACH*(op_i)))$ correspondante. Nous utilisons la relation *ioco* pour déterminer si l'implémentation est conforme à la spécification donnée. Ainsi les actions de sortie non spécifiées (y compris les blocages), rendent les implémentations testées non

conformes. Nous détaillons dans cette section la relation ioco : $IUT(op_i)\ ioco\ ACH * (op_i) \equiv \forall\sigma \in trace(T(ACH * (op_i))) : out(trace(IUT(op_i))$ after $\sigma) \subseteq out(trace(T(ACH * (op_i)))$ after $\sigma)$. Ainsi $IUT(op_i)$ est ioco-conforme à $ACH * (op_i)$ si et seulement si :

- si $IUT(op_i)$ produit la sortie x après la trace σ, alors $T(ACH * (op_i))$ peut produire x après σ.
- si $IUT(op_i)$ ne peut pas produire de sortie après la trace σ, alors $T(ACH * (op_i))$ ne peut pas produire de sortie après σ.

Selon la relation *ioco*, l'implémentation sous test est conforme avec l'arbre de chemins, alors l'opération est robuste. Une implémentation est non conforme si une exécution d'un cas de test $TC \in T$ comporte au moins une action de sortie non spécifiée. Donc, dans ce cas l'opération n'est pas robuste. Nous définissons les verdicts suivants :

- *pass* pour les états appartenant à une séquence d'exécution σ de l'IUT conforme à $ACH*$,
- $fail$ pour les états appartenant à une séquence d'exécution σ de l'IUT non conforme à $ACH*$.

4 Conclusion

Dans un service Web composé, plusieurs opérations dépendent les unes des autres. Afin d'optimiser notre méthodologie de test, nous considérons qu'une opération non robuste implique que toute opération qui passe par la branche de cette opération est non robuste. De ce fait, nous commençons à tester les opérations selon leur ordre de réponse. Puisque la première opération qui fournit une réponse est celle à laquelle dépendent les autres opérations. Nous avons proposé une approche générique axée sur le test de services Web composés. Pour la suite, nous envisageons de prendre en considération la même plate-forme pour tester la sécurité des services Web composés en appliquant des règles de sécurité sur les arbres de chemins symboliques.

Références

[1] L. Bentakouk, P. Poizat, and F. Zaidi. A formal framework for service orchestration testing based on symbolic transition systems. In *21th IFIP- TestCom/FATES*, 2009.

[2] M. Bozkurt and M. Harman. Optimised realistic test input generation. In *The 3rd International Symposium on Search Based Software Engineering*, 2011.

[3] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *8th IFIP TC 6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*, volume 3964, New York, USA, May 2006.

[4] M. Hennessy and H. Lin. Symbolic bisimulations. In *Theoretical Computer Science*, 1995.

[5] S.H. Kuk and H.S. Kim. Robustness testing framework for web services composition. In *4th IEEE Asia-Pacific Services Computing Conference (APSCC)*, 2009.

[6] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang. Automatic timed test case generation for web services composition. In *6th IEEE European Conference on Web Services (ECOWS)*, 2008.

[7] S. Salva and I. Rabhi. Automatic web service robustness testing from wsdl descriptions. In *12th European Workshop on Dependable Computing*, Toulouse, Fance, May 2009.

[8] J. Tretmans. Conformance testing with labelled transition systems : Implementation relations and test generation. *Computer Networks and ISDN Systems*, pages 49–79, 1996.

[9] M. Vieira, N. Laranjeiro, and H. Madeira. Assessing robustness of web-services infrastructures. *Int. Conf. On Dependable Systems and Networks (DSN)*, 2007.

[10] J. L. Zhong and S. Wei. Bpel-unit : Junit for bpel processes. In *SERVICE-ORIENTED COMPUTING-ICSOC*, 2006.