Une Approche de Test de Charge de Compositions de Services Web

Afef Jmal Maâlej*, Moez Krichen**

*Laboratoire ReDCAD, Université de Sfax, Tunisie
afef.jmal@redcad.org
www.redcad.org/members/afef.jmal
**Faculté de CSIT, Université Al-Baha, Arabie Saoudite
Laboratoire ReDCAD, Université de Sfax, Tunisie
moez.krichen@redcad.org
www.redcad.org/members/mkrichen

Résumé. Nous nous intéressons dans ce papier au test à base de modèles de services Web composés dans le but d'étudier leurs limitations en particulier sous des conditions de charge variées. Dans le but de mettre en évidence notre solution, nous avons suivi une démarche de test incrémentale commençant par le test de conformité d'une instance de composition de services Web. Ensuite, nous avons étudié le test de charge de ces applications d'une manière significative. Pour ce faire, nous avons réalisé la surveillance de ces applications durant le test de charge, pour ensuite effectuer une étape d'analyse automatisée des résultats de test visant, en particulier, à identifier les causes ainsi que les natures des éventuels problèmes.

1 Introduction

Vu l'émergence de l'utilisation des compositions de services Web et vu qu'elles doivent fournir des services à des centaines d'utilisateurs simultanément, le test de charge Beizer (1990) de telles applications constitue une tâche importante afin d'assurer leur qualité de service et d'identifier les problèmes liés à la montée en charge.

En effet, notre approche de test permet aux testeurs d'étudier les limitations des compositions de services Web sous charge dans leur contexte d'exécution Maâlej et Krichen (2015). En effet, nous pouvons identifier dans notre travail si le problème détecté sous charge est provoqué par des anomalies d'implémentation (comportements non spécifiés ou délais erronés), des problèmes au niveau de l'environnement de test (communication avec un service partenaire) ou au niveau du nœud du SUT (retard de traitement). Dans le but de mettre en évidence notre solution, nous suivons une démarche de test incrémentale commençant par (1) le test de conformité d'une instance de composition de services Web Maâlej et al. (2012b). Ensuite, nous nous intéressons par (2) ce même type de test mais en considérant diverses conditions de charge Maâlej et al. (2013b). Enfin, nous (3) traitons le test de charge des compositions de services Web d'une manière plus avancée et plus significative Maâlej et Krichen (2015). Pour cela, nous définissons et validons une approche automatisée basée sur l'interception des

messages échangés entre la composition sous test et ses services partenaires. De cette manière, il devient possible de surveiller les messages instantanément, et d'identifier quelle est la cause derrière leur perte ou probablement leur retard de réception, etc.

Le reste de ce papier est organisé comme suit : nous exposons dans la Section 2 un état de l'art concernant le test de charge. Les Sections 3, 4, 5 et 6 seront dédiées pour présenter l'architecture de test adoptée tout en décrivant le principe de l'approche proposée et la procédure de test. Ensuite, nous présentons les modules élaborés de monitoring (Section 7) et d'analyse (Section 8). En plus, nous détaillons les limitations des compositions de services Web détectées sous charge. La Section 9 conclut ce papier et présente quelques perspectives.

2 État de l'art

Une étude exhaustive de l'existant a montré que plusieurs travaux ont traité le test de charge dans des domaines variés tels que le test des applications distribuées Garousi et al. (2006), des applications temps-réel Briand et al. (2005) et des applications Web Wang et al. (2010); Zhou et al. (2014). La plupart des travaux existants se sont intéressés à vérifier uniquement les contraintes temporelles de l'application et reporter quelques paramètres de performance observés sous charge Garousi et al. (2006); Briand et al. (2005); Wang et al. (2010); Avritzer et Larson (1993); Vögele et al. (2014); Zhou et al. (2014). Toutefois, il est nécessaire de vérifier le comportement fonctionnel d'une application donnée sous des conditions de charge. En effet, dans quelques organismes agiles Hinds (2014), des changements se produisent assez rapidement. Il est possible à une nouvelle caractéristique ou à une certaine nouvelle fonctionnalité d'être vérifiée dans le contrôle de source, d'être exécutée sur une plate-forme d'intégration continue, de réussir tous les tests automatisés, et enfin d'être déployée sur un serveur de production dans quelques minutes. Mais, si tout ce code n'était pas optimisé pour manipuler, simultanément, un nombre important d'utilisateurs, il pourrait faire écraser le système entier. L'intégration du test de charge dans le processus avant que ces changements soient déployés peut garantir que les utilisateurs obtiennent tous les besoins attendus sans mauvaises expériences d'utilisation. Plusieurs erreurs, dans ce sens, peuvent apparaître si l'application est chargée alors qu'elles n'apparaissent pas dans des conditions d'exécution normales. Ces erreurs sont décrites comme *sensibles à la charge* Beizer (1984).

2.1 Test de charge des compositions de services Web

Concernant notre contexte de travail, à notre connaissance, aucune étude de recherche n'a abordé le test de charge significativement pour les compositions de services Web. Nous partageons un objectif avec les approches de Jiang et al. Jiang et al. (2008) et Jiang Jiang (2010), qui est l'identification des problèmes durant le test de charge en analysant systématiquement les journaux résultants des tests de charge. Cette approche consiste à identifier le comportement dominant de l'application qui sera par la suite utilisée comme référence pour détecter les erreurs. Tandis que notre approche se base sur un modèle formel et consiste à vérifier d'une part la conformité de l'application par rapport à ce modèle et donc à détecter les erreurs d'implémentation sous des conditions de charge. D'autre part, nous visons à identifier d'autres types d'erreurs ainsi que leurs causes en réalisant le monitoring de l'application durant le test de

charge Maâlej et Krichen (2015). En effet, nous pouvons identifier dans notre travail si le problème détecté sous charge est provoqué par des anomalies d'implémentation, des problèmes au niveau de l'environnement de test ou au niveau du nœud du SUT. Cependant, nous émettons l'hypothèse que les communications dans le réseau entre les nœuds de notre architecture de test sont fiables. Par conséquent, nous n'évoquons pas de problèmes au niveau du réseau. Nous soulignons que la contribution principale de notre travail est la vérification des exigences des compositions de services Web sous des conditions de charge diverses, ceci en proposant une solution rigoureuse de test fonctionnel et de charge à base de modèles Maâlej et Krichen (2016). Par ailleurs, les résultats finaux au sujet des comportements anormaux et des taux d'erreurs observées sont également fournis dans le but d'identifier et aborder les problèmes détectés sous charge. Certainement tous ces points rendent notre approche de test de charge plus riche et plus intéressante que celles existantes.

3 Aperçu général de notre solution

Tout d'abord, nous précisons que notre technique de test de charge est qualifiée d'être en boîte grise vu que nous simulons les différents services partenaires de la composition sous test. En effet, seules les interactions entre la composition et ses partenaires sont connues. De plus, notre technique de test est en ligne étant donné que les phases de génération et d'exécution des cas de test se font simultanément Mikucionis et al. (2004). En outre, notre solution permet de réaliser un test à base de modèle des compositions BPEL Barreto et al. (2007) sous des conditions de charge, ceci consiste à vérifier l'adéquation de ces applications par rapport aux exigences définies dans une spécification modélisée sous forme d'automates temporisés Maâlej et al. (2012a).

Pour cela, nous avons défini et validé une approche automatisée basée sur l'interception des messages échangés entre la composition sous test et ses services partenaires. De cette manière, il est possible de surveiller les messages instantanément, et d'identifier la cause derrière leur perte ou probablement leur retard de réception, etc. Notre approche se compose de deux étapes Maâlej et al. (2013a). La première est une étape d'exécution du test de charge et d'enregistrement du fonctionnement du processus BPEL sous test. La deuxième étape consiste à analyser les résultats observés afin de générer un rapport de test contenant les verdicts des instances invoquées d'une part, et d'identifier la nature des problèmes détectés sous charge en fournissant éventuellement leurs causes, d'autre part Maâlej et Krichen (2015).

4 Architecture de test de charge de compositions de services Web

Dans cette section, nous présentons notre architecture de test de charge de compositions de services Web décrites en BPEL Maâlej et al. (2013b).

Comme l'illustre la Figure 1, notre architecture de test de charge est composée de deux parties essentielles :

— *Les instances BPEL du service composé sous test (SUT)* : pour chaque appel de la composition sous test, une instance BPEL sera créée. Une instance BPEL est définie par un identifiant unique. Ces instances sont au nombre des appels de la composition

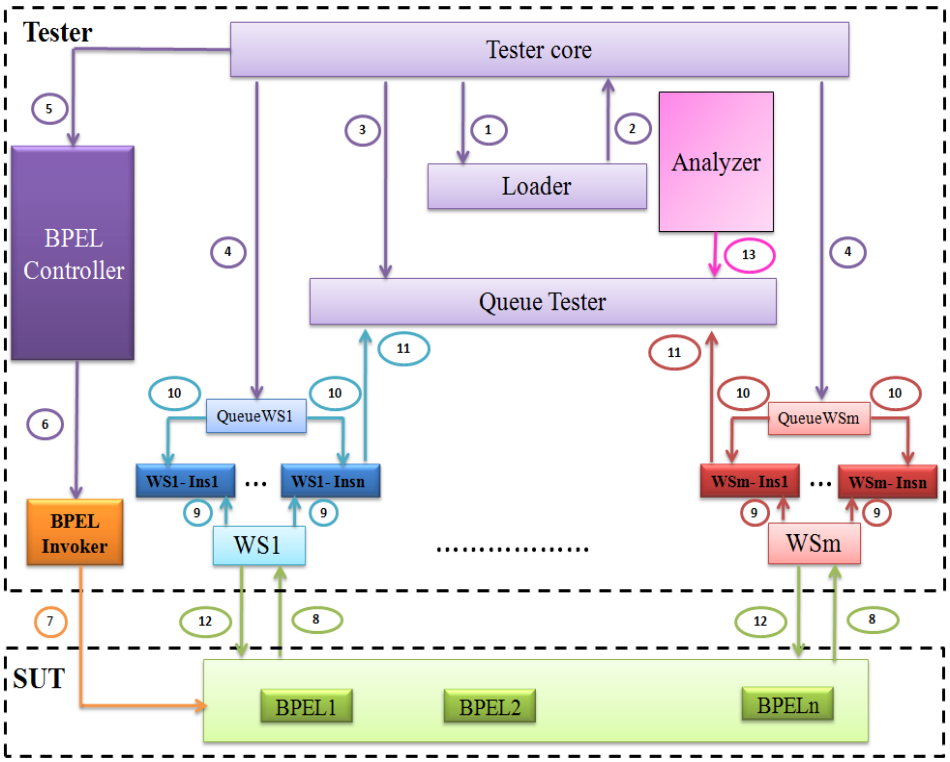


FIG. 1 – *Architecture de test de charge en ligne Maâlej et al. (2013b)*

sous test. À son tour, chaque instance BPEL créée invoque ses propres instances de services partenaires en communiquant ensemble par échange de messages.

— *Le testeur (Tester)* : il présente l'environnement du système sous test (les instances créées de la composition sous test) et il se compose :
  — *Du chargeur des documents nécessaires au test (Loader)* : il charge la spécification en automate temporisé, les fichiers WSDL et XSD de la composition sous test et les fichiers WSDL des différents services partenaires. Il implante un algorithme de parcours de la spécification et définit le type de chaque transition et état dans l'automate. D'autre part, il définit le type des variables d'entrée/sortie de la composition sous test ainsi que de ses différents services partenaires.
  — *Du contrôleur de test (Tester Core)* : il génère aléatoirement les messages d'entrée du processus BPEL ainsi que la transition à franchir lors d'une communication synchrone. Il communique avec les différents services partenaires de la composition en leur envoyant les types d'entrée et de sortie. Dans le cas de services partenaires intervenant dans des communications synchrones, il leur envoie des informations concernant le temps de réponse au service composé en se référant au chemin choisi aléatoirement. Enfin, il invoque le "BPEL Controller" pour qu'il exécute les appels

concurrents de la composition sous test.

— *Des exécuteurs de test (WS1, ..., WSm ; WS1-ins1, ..., WSn-insn ; BPEL Invoker)* : WS1, ..., WSm, BPEL Invoker sont les services partenaires simulés de la composition sous test y compris le client. À chaque appel de la composition au cours du test de charge, des instances de services partenaires seront créées (par exemple, pour WS1, les instances WS1-ins1, ..., WS1-insn seront créées). Afin d'éviter d'alourdir notre application par les communications entre le testeur et les services partenaires au cours du test, ces services interviennent dans le test. En effet, dès qu'un partenaire reçoit un message du processus BPEL, il vérifie son type en se référant au type envoyé par le testeur dans sa queue correspondante. Par la suite, il saisit les informations qu'il a reçues dans la queue du testeur. Enfin, il génère son message de sortie aléatoire en se référant au type envoyé par le testeur dans sa queue.
En particulier, le BPEL Invoker répond à l'ordre du contrôleur de test en effectuant les invocations concurrentes du service composé sous test selon les paramètres reçus (variable d'entrée du processus, nombre de threads et délai entre chaque deux invocations successives).

— *Du contrôleur d'invocation du service composé (BPEL controller)* : cette entité est responsable des invocations concurrentes du service composé sous test. Elle reçoit le nombre d'invocations requis pour le test du processus BPEL ainsi que le délai entre chaque deux invocations successives du testeur.

— *Du journal de test en ligne (QueueTester)* : il stocke les informations générales du test (nombre d'appels de la composition sous test, le délai entre les invocations des instances du BPEL, le chemin à parcourir, etc.), les identifiants des instances créées du service composé, les services invoqués, les messages qu'ils ont reçus du SUT, le temps de leurs invocations et le verdict de vérification du type des messages d'entrée des partenaires. Ce journal sera consulté par l'analyseur de test afin de vérifier le fonctionnement des différentes instances du service composé.

— *Des queues (Queue WS1, ..., Queue WSm)* : ces entités sont des services Web à travers lesquels les services partenaires et le contrôleur échangent des messages.

— *De l'analyseur de test (Analyzer)* : cette entité est responsable de l'analyse hors ligne du journal du test "Queue Tester". Elle génère un rapport final de test qui précise le verdict final de chaque instance invoquée du SUT ainsi que le taux d'erreurs observé sous une charge donnée.

5 Étapes de test de charge des compositions de services Web

Dans cette section, nos décrivons les étapes nécessaires pour effectuer le test de charge en ligne de compositions de services Web Maâlej et al. (2013a). Ces étapes sont illustrées dans la Figure 1. Une fois le testeur saisit les données nécessaires au test (la spécification décrite en automate temporisé, la description BPEL du service composé, le temps d'attente maximal du réseau *tmax*, le nombre d'appels successifs du service sous test et le délai entre deux appels) et lance le test :

1. Le contrôleur (Tester Core) appelle le chargeur des documents "Loader" afin de charger l'automate temporisé, les fichiers WSDL et XSD du service sous test et les fichiers WSDL des différents services partenaires.

2. À partir de ces fichiers, le “Loader" définit les transitions relatives à des communications synchrones. Il détermine aussi les types des variables d'entrée/sortie du service composé ainsi que des différents services partenaires. Par la suite, il envoie ces informations au contrôleur de test.
3. Suite aux informations reçues par le “Loader", le contrôleur de test saisit les informations générales de test (le nombre d'invocations concurrentes et le délai entre deux invocations successives) dans “QueueTester".
4. Le testeur envoie à chaque service partenaire les informations relatives aux types des messages d'entrée/sortie. Dans le cas des services intervenant aux communications synchrones, il leur envoie des informations nécessaires au temps de réponse au service composé.
5. Le contrôleur de test invoque le contrôleur du BPEL “BPEL Controller" avec le nombre d'appels du service composé (N) ainsi que la durée entre chaque deux invocations successives (d).
6. Suite à ces données, le “BPEL Controller" appelle le client du BPEL “BPEL Invoker" N fois chaque d secondes.
7. De sa part, le “BPEL Invoker" invoque le service composé N fois chaque durée d.
8. Les instances du service composé (BPEL1, BPEL2,..., BPELn) fonctionnent simultanément.
9. Chaque instance du service composé invoque ses propres instances de services partenaires.
10. Chaque instance du service partenaire détermine, à partir de sa queue correspondante, le type de ses variables d'entrée et de sortie et vérifie le type du message qu'elle a reçu du BPEL.
11. Chaque instance du service partenaire invoqué saisit, dans “Queue Tester", l'identifiant de l'instance du processus BPEL qui l'a invoqué, les informations qui lui sont envoyées et le résultat de la vérification du type des messages d'entrée.
12. En se référant au type de la variable de sortie dans sa queue, l'instance du service partenaire génère aléatoirement une réponse et l'envoie au SUT.
13. À la fin du test, l'analyseur de test analyse les informations stockées dans “Queue Tester" afin de générer un rapport final de test contenant les verdicts relatifs à chaque instance invoquée du service composé, ainsi que le taux d'erreur relatif à la composition sous test sous une charge donnée.

6 Algorithme de test de charge

Afin de réaliser les étapes de test de charge, nous avons développé un algorithme qui est implanté au niveau de l'entité (Tester Core) figurant dans l'architecture de test de charge illustrée dans la Figure 1 Maâlej et al. (2013b). Cet algorithme possède comme entrée une spécification formelle d'un service composé décrite en automate temporisé (AT), le fichier BPEL de description du service composé, le nombre d'appels de la composition sous test (Theads number), le délai maximal du réseau (tmax) et la durée entre chaque deux invocations successives du

SUT (Delay). À la fin, un journal de test est généré (QueueTester). Ce journal contient des informations générales du test (Delay, Threads number, tmax), les chemins possibles et les différentes actions réalisées lors du test de charge telles que les identifiants des instances de la composition sous test, les services partenaires invoqués, les messages échangés, les verdicts de vérification des types de messages, etc.

À partir de la spécification (AT), le “Loader" génère la liste des états intervenants dans les communications synchrones (synCommunications). Pour chaque état de cette liste, le contrôleur de test choisit aléatoirement une transition à franchir et définit la liste des chemins possibles selon ce choix. Par la suite, il envoie à chaque service partenaire les types de ses variables d'entrée et de sortie. Pour le cas des services partenaires participants à une communication synchrone, le testeur leur envoie les infomations relatives au temps de réponse du service partenaire au service composé. En effet, si la transition choisie par le contrôleur de test est de type “GuardInputTransition" (par exemple $S?resp\_SW, c < 60S'$), le contrôleur informe le service partenaire qu'il doit répondre au testeur dans une durée inférieure à la valeur maximale de la condition de test (pour l'exemple précédent, le service partenaire SW doit répondre à la composition dans une durée inférieure à 60 unités de temps). Si la transition choisie par le contrôleur de test est de type “TimedTransition" (par exemple $Sc = 60S'$), le contrôleur informe le service partenaire qu'il doit répondre au testeur dans une durée supérieure ou égale à la valeur maximale de la condition de test (pour l'exemple précédent, le service partenaire SW doit répondre à la composition dans une durée supérieure ou égale à 60 unités de temps). Enfin, l'entité “BPEL Controller" appelle le “BPEL Invoker" qui invoque la composition sous test autant de fois que la valeur de “Threads_number" indiqué par le testeur.

7 Module de monitoring

Au cours du test de charge, la composition BPEL sous test est supervisée et des journaux relatifs à son exécution et aux données de performance sont stockées. Les services Web partenaires jouent un rôle important dans la journalisation. En effet, une méthode simple de collecte des informations (temps d'invocation, temps de réponse) consiste à insérer des fonctionnalités dans le code client de ces services.

D'autre part, le test de charge est accompagné d'une surveillance de performance de l'environnement d'exécution pour pouvoir superviser toute l'infrastructure système pendant le test. En fait, l'utilisation d'un module de monitoring de performance permet de sélectionner avant le début du test les grandeurs à surveiller (utilisation processeur, utilisation du disque, etc.) et permet d'afficher l'évolution en temps réel de ces métriques. En outre, ce module de monitoring aide à déterminer la corrélation entre les problèmes de performance et les erreurs détectées par la plate-forme de test en superposant par exemple une courbe d'évolution du temps d'exécution d'un processus BPEL sous charge et une courbe illustrant les valeurs observées d'un critère de performance donné. Ce suivi de performance est facilité par un certain nombre d'outils destinés pour la collecte de façon continue de paramètres spécifiques et la notification des changements critiques. Dans notre travail, nous avons utilisé l'outil de monitoring système PerfMon [1] sous Windows afin de collecter périodiquement différentes métriques de performance, et de géné-

1. http ://perfmon.sourceforge.net/

rer des journaux pouvant être analysés ensuite pour l'identification des problèmes relatifs à la performance.

Par conséquent, cette phase de monitoring consiste à collecter et produire des informations diverses afin de les fournir aux étapes suivantes, en particulier, à l'étape d'analyse.

8 Module d'analyse

La détection des problèmes dans un test de charge est une tâche difficile. En fait, le test de charge est généralement parmi les dernières étapes dans le cycle de développement d'un système. Aussi, le temps alloué pour analyser les résultats de test est limité. D'autre part, le test de charge enregistre un grand volume de données et de journaux qui doivent être analysés de façon approfondie afin de découvrir les problèmes suite au test. Pour cela, l'analyse manuelle du test de charge n'est pas efficace. Pourtant, peu d'efforts de recherche sont dédiés pour l'analyse automatisée des résultats de test de charge, généralement dû à l'accès limité aux systèmes à large échelle utilisés comme des études de cas. L'analyse de test de charge automatisée et systématique devient de plus en plus nécessaire, comme plusieurs services sont offerts en ligne pour un nombre croissant d'utilisateurs.

Par motivation de l'importance et des défis de l'analyse des tests de charge, nous proposons dans ce qui suit une approche automatisée pour la détection de problèmes fonctionnels et qui sont liés à la performance, suite à l'exécution d'un test de charge d'une composition de services Web, ceci en analysant les journaux déjà enregistrés au cours de la phase de test, ainsi que des métriques de performance observées.

8.1 Principe de l'approche d'analyse des tests de charge

Dans cette section, nous présentons notre approche automatisée pour la détection des problèmes liés au test de charge pour les compositions de services Web Maâlej et al. (2013b).

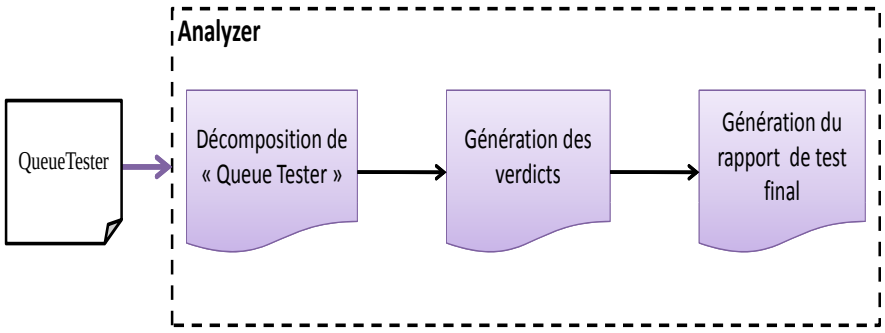


FIG. 2 – *Technique automatisée d'analyse du journal de test "QueueTester" Maâlej et al. (2013b)*

Comme le montre la Figure 2, cette approche prend comme entrée les informations stockées dans “Queue Tester" durant le test de charge. Elle passe par trois étapes : (1) décomposition des informations de “Queue tester", (2) vérification du fonctionnement de chaque instance créée et enfin, (3) génération du verdict correspondant et génération du rapport final de test. Dans ce qui suit, nous détaillons ces étapes.

— *Décomposition de “QueueTester"* : les opérations réalisées au cours du test de charge de compositions de services Web sont stockées dans “QueueTester". Afin de distinguer l’instance du processus BPEL responsable d’une opération donnée, chaque opération commence par l’identifiant de l’instance du processus BPEL qui lui est responsable (BPEL-ID). À la fin de l’exécution du test, l’analyseur de test consulte “QueueTester". En se basant sur BPEL-ID, il décompose les informations dans des rapports de test. Chaque rapport portera le nom de BPEL-ID et contiendra les informations relatives à l’instance dont l’identifiant est BPEL-ID.

— *Analyse des rapports et génération des verdicts* : l’analyseur de test consulte les rapports des différentes instances du service composé sous test. Il vérifie le fonctionnement de chaque instance en se référant à la liste des chemins possibles générée par Tester Core. Enfin, il génère un verdict de test final pour chaque instance.
Un verdict “FAIL" est généré dans les cas suivants :
  — Si le nombre des rapports générés suite à la décomposition de “QueueTester" est inférieur à “Threads number" inscrit dans “QueueTester".
  — Si le rapport d’une instance BPEL créée contient déjà un verdict “FAIL" généré suite à la vérification des types de messages échangés entre les instances BPEL et les différents services partenaires ou la vérification de la réponse finale de l’instance du processus BPEL.
  — Si le chemin parcouru par l’instance BPEL du rapport courant n’est pas conforme à l’un des chemins possibles décrits dans la section “Possible Paths" du journal de test “QueueTester".

  Si aucun de ces cas apparaît, un verdict “PASS" est généré.

— *Génération du rapport de test final* : à la fin de l’analyse, l’analyseur génère un rapport final de test contenant le verdict correspondant à chaque instance et le taux d’erreur de la composition sous test à une charge donnée. Le taux d’erreur se calcule par la formule suivante :

$$tauxd'erreur = \frac{nombred'instanceserrones}{nombretotald'instances} \times 100$$

8.2 Classification des problèmes détectés sous charge

Dans notre travail, nous émettons l’hypothèse que les communications dans le réseau entre les nœuds de notre architecture de test sont fiables. Par conséquent, nous n’évoquons pas de problèmes au niveau du réseau.

Dans cette partie, nous présentons et nous classifions les problèmes notés par expérimentations durant l’exécution des tests de charge pour différentes compositions de services Web Maâlej et Krichen (2015). En particulier, nous allons discuter trois sources de ces problèmes.

8.2.1 Problèmes fonctionnels (SUT)

Les erreurs sensibles à la charge dans les programmes peuvent n'avoir aucun effet endommageant sous des charges petites ou au cours des exécutions courtes, mais peuvent causer l'échec d'un programme lorsqu'il est exécuté sous des charges importantes ou sur une période de temps étendue, ce qui peut résulter en des non-conformités par rapport à la spécification. Dans notre contexte, nous considérons essentiellement deux fautes sensibles à la charge dans l'implémentation du SUT :

Comportements non spécifiés
Cette erreur signifie qu'un comportement non spécifié est ajouté (ou omis) dans une branche du service composé se traduisant par la réalisation de traitements non requis (ou l'absence de fonctionnalités attendues). Cette erreur peut figurer au niveau d'une branche conditionnelle (contrainte temporelle définie par l'activité "pick" ou condition logique définie par l'activité "switch"). En effet, la charge peut influencer la variation des temps de réponse d'un service partenaire. De plus, elle peut affecter la décision du choix des chemins à suivre, contrôlant ainsi l'exécution du flux BPEL.

La Figure 3 illustre un ajout de comportement non spécifié au niveau d'une contrainte temporelle définie par l'activité "pick". Dans cet exemple simplifié de composition, nous supposons que la réception d'une réponse du service A (WSA) est conditionnée par un délai maximum égal à 60 secondes, comme spécifié en automate temporisé dans la Figure 3. Selon le temps de réponse de ce service, soit la branche de droite est suivie ($c < 60$) et le service (WSB) est invoqué, soit la branche de gauche est choisie ($c >= 60$) et le processus se termine. Cependant, l'implémentation présentée à droite de cette Figure comporte une erreur qui peut être détectée sous charge. En effet, il s'agit de l'invocation non spécifiée du service (WSC) en cas de dépassement du délai prévu (60 secondes) de la réponse du service (WSA).

Effectivement, la charge peut influencer la variation du délai de réponse d'un service partenaire et agit sur le choix de la branche suivie. Pour une faible charge, les instances BPEL concurrentes suivent la branche de droite. En augmentant de plus en plus la charge, le serveur d'orchestration déployant la composition sous test se charge davantage et nous remarquons que certaines instances suivent la branche de droite et d'autres suivent celle de gauche. C'est à travers ces dernières instances qui exécutent le chemin de gauche que nous pouvons détecter cette erreur de codage.

De plus, nous illustrons aussi le cas d'une omission de comportement requis à travers la Figure 4. Cette omission se situe au sein d'une branche soumise à une condition logique définie par l'activité "switch".

En effet, d'après la spécification présentée à gauche de cette Figure, la réponse "output" du service (WSA) détermine le choix de la branche à suivre. Si "output" est égale à la valeur "val" (output==val), le service composé poursuit son chemin en invoquant le service (WSB), sinon, le service (WSC) est invoqué. Tandis que dans l'implémentation (présentée à droite), l'invocation du service (WSC) est manquante en cas de réponse non satisfaisante (output !=val) du service (WSA). Cette omission de fonctionnalité attendue peut être détectée avec l'augmentation de la charge lorsqu'il s'agit d'une réponse non satisfaisante à cause de ressources insuffisantes, stocks manquants, etc. Ainsi, la charge exercée sur une composition donnée peut influencer une condition à vérifier au sein du flux BPEL.

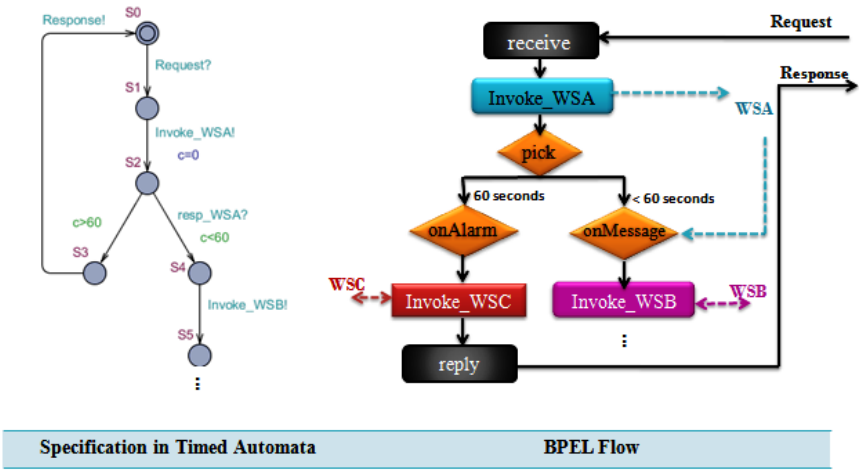


FIG. 3 – *Illustration d'un exemple d'ajout d'un comportement non spécifié*

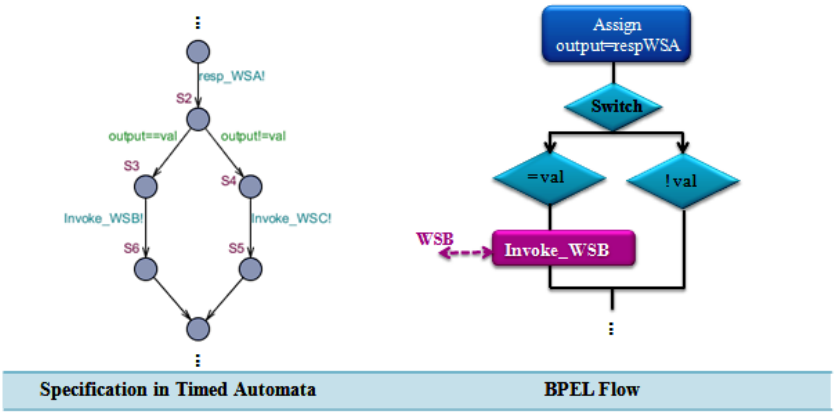


FIG. 4 – *Illustration d'un exemple d'omission d'un comportement requis*

Délais erronés
Cette erreur peut figurer au niveau d'une activité *"pick"* de la composition sous test. Elle consiste en une implémentation de communication synchrone conditionnée par un délai d'attente de réponse d'un service partenaire différent de celui spécifié dans l'automate temporisé. Afin d'illustrer ce type d'erreur, nous considérons une composition sous test dans laquelle la réception d'une réponse du service (WSA) est conditionnée par un délai maximum égal à 60 secondes. Selon le temps de réponse de ce service, soit la branche *«onMessage»* est suivie (c $<$ 60), soit la branche *«onAlarm»* est choisie (c $>=$ 60).

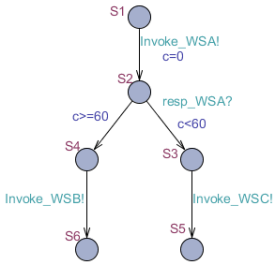
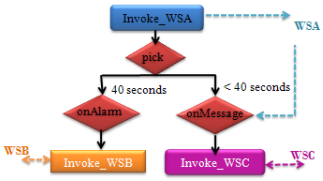
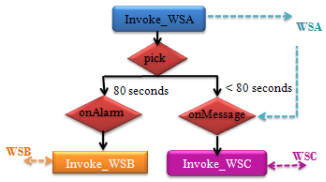
Pour clarifier la détection de ce type d'erreurs, nous supposons que le codeur de cette composition a commis une erreur en fixant comme délai d'attente une valeur *différente* de 60 secondes (soit plus petite, soit plus grande). Dans ce cas, deux illustrations du problème sont observées dans le Tableau 1 Maâlej et Krichen (2015).

Selon ces observations, nous notons que la variation des temps de réponse (t) du service (WSA) influence la conformité de l'implémentation (I) par rapport à la spécification (S). En effet, le comportement observé dans (I) est conforme à celui requis dans (S) si :
$t < min(dI, dS)$ ou si : $t > max\ (dI,\ dS)$. Tandis que (I) n'est pas conforme à (S) si $min\ (dI,\ dS) < t < max\ (dI,\ dS)$. Il convient de préciser que la variation du temps (t) reflète le changement du temps de réponse du service (WSA), qui dépend essentiellement de la variation de la charge exercée sur ce service.

En outre, les deux cas correspondant à $dI = 40s$ et $dI = 80s$ contrarient le délai d'attente spécifié ($dS = 60s$). Malgré ceci et d'après le Tableau 1, pour des temps de réponse $t < min\ (dI, dS)$ (petites charges) ou bien $t > max\ (dI, dS)$ (charges importantes), ces délais erronés dI ne sont pas détectés. Par contre, pour des temps de réponse t entre ces deux seuils (charges modérées), il y a clairement une non-conformité entre (I) et (S). En effet, les branches suivies réellement dans l'implémentation sont différentes de celles spécifiées.

— Interprétation de la variation des temps de réponse t pour un service partenaire donné :
En général, un service partenaire d'une composition BPEL donnée réalise un traitement local et/ou requiert une ressource externe (fichier, base de données, etc.) dans le but de répondre à la requête d'un client. Cependant, si la charge exercée sur ce service augmente, alors le temps d'exécution ou/et le temps d'accès aux ressources externes (lecture/écriture) va augmenter, ce qui retarde la réponse attendue par le processus. Par conséquent, le temps de réponse t augmentera aussi [2].
En outre, la source de la charge à laquelle est sujet un service partenaire d'une composition donnée peut être soit cette composition soit des entités externes à ce service. Dans notre cas, afin de contrôler et varier le temps de réponse d'un service partenaire quelconque, nous chargeons notre composition sous test en stressant le serveur d'orchestration BPEL. De cette façon, plusieurs services partenaires de la composition sous test vont être affectés par cette charge, ce qui cause une variation de leurs temps de réponse. Ainsi, nous observons dans ce cas différentes branches suivies par l'implémentation BPEL (selon la valeur de t comme illustré dans le Tableau 1). Selon l'intervalle auquel appartient t , une erreur fonctionnelle sensible à la charge pourra être révélée.

2. Nous avons prouvé ce comportement par expérimentation pour des services partenaires ayant différentes portées (scopes) de déploiement (en particulier pour les portées requête, application et session). En effet, ces portées ont montré le même comportement envers la charge appliquée aux services correspondants.

Spécification (S) d'une communication synchrone avec le service (WSA)	**Illustration d'implémentations (I) erronées du délai d'attente (d) d'une réponse du service (WSA)**	
	Délai implémenté<Délai spécifié (40 secondes) $(60 seconds)$	Délai implémenté>Délai spécifié (80 secondes) $(60 seconds)$
		
Suivi du comportement de l'implémentation BPEL (I) par rapport à la spécification (S) selon la valeur du temps de réponse (t) observé du service (WSA) dI : délai implémenté (40 ou 80 secondes) dS : délai spécifié (60 secondes)		
t < min (dI,dS)	Pour t < 40s : Branche «**onMessage**» est suivie par I : Comportement conforme à la spécification S	Pour t < 60s : Branche **«onMessage»** est suivie par I : Comportement conforme à la spécification S
min(dI,dS)<t<max(dI,dS)	Pour 40s<t<60s : Branche **«onAlarm»** est suivie par I : Comportement non conforme à la spécification S	Pour 60s<t<80s : Branche **«onMessage»** est suivie par I : Comportement non conforme à la spécification S
t > max (dI,dS)	Pour t > 60s : Branche **«onAlarm»** est suivie par I : Comportement conforme à la spécification S	Pour t > 80s : Branche **«onAlarm»** est suivie par I : Comportement conforme à la spécification S

TAB. 1 – *Illustration d'exemples de délais erronés Maâlej et Krichen (2015)*

Classification des problèmes fonctionnels

Comme déjà expliqué, nous avons défini trois erreurs au niveau du codage d'une composition

sous test qui peuvent être détectées en stressant le serveur déployant cette composition. Nous pouvons classifier ces erreurs comme suit :

— Erreurs sémantiques à impact total sur la conformité :
Il s'agit en fait de (1) l'ajout de comportement(s) non spécifié(s) dans une branche conditionnelle, ou encore (2) l'omission (oubli) de comportement(s) requis dans une branche conditionnelle. En effet, ces deux types d'erreurs touchent la sémantique de la composition sous test. Une fois ces erreurs sont rencontrées dans une branche donnée, alors l'implémentation BPEL est toujours non conforme à sa spécification.

— Erreurs syntaxiques à impact partiel sur la conformité :
Il s'agit de délai d'attente implémenté d'une façon erronée. Dans ce cas, ce délai est codé syntaxiquement différemment de ce qui est spécifié ($dI < dS$ ou bien $dI > dS$). Cette erreur influe (partiellement) sur la conformité entre (I) et (S) selon l'intervalle de temps de réponse observé du service partenaire. En effet, (I) est conforme à (S) lorsque $t < \min(dI,dS)$ ou bien $t > \max(dI,dS)$. Tandis que I est non conforme à (S) lorsque $\min(dI,dS) < t < \max(dI,dS)$.

Classification des activités BPEL vis-à-vis de la charge

D'après l'étude que nous avons menée, nous distinguons les classifications suivantes :

— Les activités BPEL sensibles à la charge :
À leur niveau, des erreurs de codage sont détectées sous charge. Il s'agit des activités *pick* et *switch*. En effet, ces deux activités contiennent des branches conditionnelles dont chacune implémente un comportement donné différent et dont le choix est déterminé selon la condition correspondante.

— Les activités BPEL non sensibles à la charge :
Les erreurs d'implémentation au niveau de ces activités sont détectées directement avec une seule instance BPEL. Ce n'est pas la peine de charger le serveur pour les déceler (*invoke* : détermine le service partenaire requis quelque soit la charge, *wait* : définit un délai d'attente avant d'effectuer un traitement donné aussi indépendamment de la charge, etc.).

8.2.2 Environnement de test

En plus de la considération de l'aspect fonctionnel de l'application, nous visons aussi à tenir compte de son contexte d'exécution et à identifier les éventuels problèmes non fonctionnels. En fait, les services Web partenaires, les serveurs d'application les déployant et les nœuds de l'architecture de test peuvent influencer l'exécution du test de charge et engendrer parfois des erreurs.

Une composition de services Web est un flux d'interactions entre un service central et ses partenaires. Lors de cette interaction, des problèmes liés aux services partenaires et/ou aux serveurs d'application peuvent être observés. Nous disons que ces problèmes sont dus à l'environnement du test. Pour déterminer à quel niveau de la communication l'erreur s'est produite, nous distinguons deux types d'erreurs :

a. Problème de connexion à un service partenaire :

Le processus BPEL n'arrive pas à invoquer un service partenaire ce qui stoppe généralement son exécution. En fait, la disponibilité d'un service Web est influencée par la charge, par l'état du serveur sur lequel il est déployé, etc.

b. Problème de récupération de réponse d'un service partenaire :

Le processus BPEL ne reçoit pas une réponse de la part d'un service partenaire invoqué. En fait, pour éviter une longue durée d'attente d'une réponse d'un service partenaire, cette durée est limitée par un délai d'attente maximal du réseau noté *tmax*. Ainsi, le service composé attend un délai maximal de *tmax* secondes pour recevoir la réponse de n'importe quel service partenaire comme le montre la Figure 5.

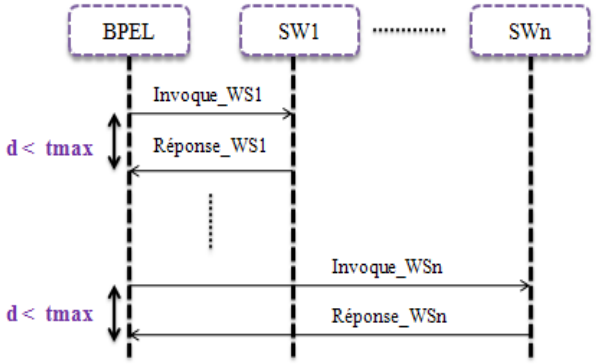


FIG. 5 – *Communication entre un processus BPEL et ses partenaires*

8.2.3 Nœud du SUT

Comme déjà illustré précédemment, l'application et l'environnement de test peuvent être des sources de problèmes lors de l'exécution du test de charge. Dans d'autres situations, nous pouvons juste prédire le nœud (la machine du testeur ou la machine du SUT) qui a causé des problèmes. Par expérimentations, nous avons observé dans quelques scénarios, au niveau du nœud SUT, un retard de traitement de la réponse d'un service. Il s'agit en effet d'un service partenaire qui envoie sa réponse à la composition sous test, au sein d'une communication synchrone, avant le délai maximum spécifié, sauf que la composition concernée suit la branche **«onAlarm»**. Cette situation peut être expliquée par le fait que les instances BPEL s'exécutant en parallèle, partagent les mêmes ressources de la machine du SUT telles que le processeur, la mémoire, etc., ce qui engendre parfois un retard de traitement au niveau de quelques instances.

8.2.4 Synthèse

Le Tableau 2 récapitule les différents types d'erreurs déjà expliquées et qui sont considérées dans notre étude des limitations des compositions BPEL sous des conditions de charge variées Maâlej et Krichen (2015).

Nature du problème	Cause du problème
— Comportements non spécifiés — Délais erronés	Application
— Problème de connexion à un service partenaire — Problème de récupération de réponse d'un service partenaire	Environnement de test
— Retard de traitement de réponse d'un service partenaire	Nœud SUT

TAB. 2 – *Classification des problèmes rencontrés sous charge Maâlej et Krichen (2015)*

8.3 Algorithme de détection des problèmes sous charge

Afin de détecter les limitations des compositions de services Web suite à l'exécution du test de charge, nous avons développé un algorithme qui est implanté au niveau de l'entité "Analyzer" figurant dans l'architecture de test Maâlej et Krichen (2015). Cet algorithme permet, entre autres, d'attribuer à chaque instance exécutée pendant le test de charge un verdict qui peut être :

— *PASS* : le test s'est exécuté correctement.
— *FAIL* : le test a échoué.
— *INCONCLUSIVE* : le test n'a pas échoué, mais nous ne sommes pas certains qu'il a réussi. Il s'agit d'une situation non décisive.

9 Conclusion

Dans ce papier, nous avons présenté et expliqué notre solution de test de charge dans le contexte des compositions de services Web à travers la proposition d'une architecture de test, ainsi que la description de notre technique d'analyse automatisée des journaux générés suite au test.

Pour l'amélioration de notre approche, nous prévoyons à court terme de conforter l'utilisateur à travers l'automatisation de la création et du déploiement des services Web simulés, faire évoluer ces architectures pour le traitement des activités parallèles au sein des processus BPEL et implémenter de nouvelles stratégies d'exploration de chemins de la spécification. En outre, pour améliorer le test fonctionnel sous charge, nous envisageons de faire des analyses plus approfondies en particulier concernant les types de fautes sensibles à la charge au sein d'une implémentation BPEL.

Références

Avritzer, A. et B. Larson (1993). Load testing software using deterministic state testing. In *ACM SIGSOFT Software Engineering Notes*, Volume 18, pp. 82–88. ACM.

Barreto, C., V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, D. Knig, S. Moser, R. Stout, R. Ten-Hove, I. Trickovic, D. van der Rijn, et A. Yiu (2007). *Web Services Business Process Execution Language Version 2.0 Primer*. OASIS.

Beizer, B. (1984). Software system testing and quality assurance. *Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, New York : Van Nostrand 1*.

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. New York, NY, USA : Van Nostrand Reinhold Co.

Briand, L. C., Y. Labiche, et M. Shousha (2005). Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, New York, NY, USA, pp. 1021–1028. ACM.

Garousi, V., L. C. Briand, et Y. Labiche (2006). Traffic-aware stress testing of distributed systems based on uml models. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, pp. 391–400. ACM.

Hinds, T. (2014). Combining agile with load and performance testing : What am i in for? `http://www.neotys.com/blog/combining-agile-load-performance-testing/`.

Jiang, Z. M. (2010). Automated analysis of load testing results. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, pp. 143–146. ACM.

Jiang, Z. M., A. E. Hassan, G. Hamann, et P. Flora (2008). Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, Beijing, China, pp. 307–316. IEEE.

Maâlej, A. J., M. Hamza, et M. Krichen (2013a). Wsclt : A tool for ws-bpel compositions load testing. In *Proceedings of the 22nd IEEE International Conference on Enabling Technologies : Infrastructures for Collaborative Enterprises (WETICE)*, Hammamet, Tunisia. IEEE Computer Society.

Maâlej, A. J., M. Hamza, M. Krichen, et M. Jmaïel (2013b). Automated significant load testing for ws-bpel compositions. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Luxembourg, pp. 144–153. IEEE Computer Society.

Maâlej, A. J. et M. Krichen (2015). Study on the limitations of WS-BPEL compositions under load conditions. *Comput. J. 58*(3), 385–402.

Maâlej, A. J. et M. Krichen (2016). A model based approach to combine load and functional tests for service oriented architectures. In *Proceedings of the 10th International Workshop on Verification and Evaluation of Computer and Communication System (VECoS)*, Volume 1689 of *CEUR Workshop Proceedings*, Tunis, Tunisia, pp. 123–140. CEUR-WS.org.

Maâlej, A. J., M. Krichen, et M. Jmaïel (2012a). Conformance testing of ws-bpel compositions under various load conditions. In *Proceedings of the 36th IEEE Annual International*

Computer Software and Applications Conference, Izmir, Turkey, pp. 371. IEEE Computer Society.

Maâlej, A. J., M. Krichen, et M. Jmaïel (2012b). Model-based conformance testing of ws-bpel compositions. In *Proceedings of the 36th IEEE Annual International Computer Software and Applications Conference Workshops*, Izmir, Turkey, pp. 452–457. IEEE Computer Society.

Mikucionis, M., K. G. Larsen, et B. Nielsen (2004). T-uppaal : Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, pp. 396–397. IEEE Computer Society.

Vögele, C., A. Brunnert, A. Danciu, D. Tertilt, et H. Krcmar (2014). Using performance models to support load testing in a large soa environment. In *Proceedings of the 3rd International Workshop on Large Scale Testing*, New York, NY, USA, pp. 5–6. ACM.

Wang, X., B. Zhou, et W. Li (2010). Model based load testing of web applications. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 483–490. IEEE.

Zhou, J., B. Zhou, et S. Li (2014). LTF : A model-based load testing framework for web applications. In *Proceedings of the 14th International Conference on Quality Software*, Allen, TX, USA, pp. 154–163. IEEE.

Summary

We are interested in this paper in the model-based testing of composed Web services with the aim to study their limitations particularly under various load conditions. In order to realize our solution, we followed an incremental test approach starting with the conformance testing of an instance of a Web services composition. Then, we studied the load testing in a significant way. With this intention, we carried out the monitoring of these applications during load testing, for after that carrying out an automated analysis of test results aiming, particularly, to identify the causes as well as the natures of possible problems.