Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1314-1323
DOI: 10.23940/ijpe.18.06.p23.13141323

Fault Injection for Performance Testing of Composite Web Services

Ju Qian*, Han Wu, Hao Chen, Changjian Li, and Weiwei Li

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China

Abstract

Fault injection has already been used to access the dependability of web services. However, most of the existing work focuses on how to inject faults. Problems such as where to inject faults and what faults should be injected still have not been systematically studied in literature, especially for the testing of performance related issues in composite web services. This paper presents an approach that defines coverage criteria to guide fault injection testing of performance related issues in composite web services. We generate fault injection configurations that follows the defined test criteria for systematic fault injection. The configurations specify where to inject faults and what faults should be injected, and the injected faults (e.g. message delays) are generated according to the characteristics of each individual sub-service in order to make the faults more realistic. With the fault injection configurations, the fault injection process can be automatically conducted and the performance of a composite service can be effectively evaluated.

Keywords: web service; fault injection; performance

(Submitted on March 8, 2018; Revised on April 16, 2018; Accepted on May 20, 2018)

1. Introduction

Web service is a new software paradigm that shares business logic, data, and processes across a network. With web services, a user can access functionalities via interacting SOAP packages (traditional SOAP-based web services) or JSON packages (RESTful micro services) over networks without needing to install and configure software applications locally. Because web services can add flexibility to software applications and make maintenance easy, they are intensively used in modern software systems, including business-centric information systems and embedded CPS systems [10].

Ensuring the quality of web services is an import issue for building service-based systems. Nowadays, large efforts have already been devoted to the testing of web services [2, 6]. For business logic related software quality problems, usually one can carefully design service inputs and trigger the executions of web services under such inputs to reveal faults. It is very difficult for faults to be discovered with the normal testing approaches since an unstable situation does not always manifest in a test run. Although a tester can repeatedly execute a web service, depending on luckiness to catch occasionally occurring unstable situations, and watch how the web service system handle them to validate whether the web service can always produce expected behavior, that can be very costly. This is because unstable situations often occur at very little frequency. Using the normal testing, we may not trigger these situations even after calling the service for thousands of times.

A more feasible solution to the above problem is using fault injection [16] to test such stability related problems. Fault injection injects anomalies into a system to assess the dependability of the system. Traditional fault injection approaches for web services mostly focus on functionality related issues, e.g., injecting mutated data into a system to see whether the system can handle data corruption caused by network transmission errors [4, 9, 11, 18, 21]. Besides functionality related issues, performance related issues are also very important. Compared with traditional applications, in web services, there are lots of factors that may affect the performance of a web service, and performance issues may affect the usability of a web service. Understanding performance attributes like max response time, deviation of response time, successful invocation rate, and etc. of a service under extreme situations can help service providers to further tune the service before its final deployment and make the service suitable for high-quality requirement application scenarios.

* Corresponding author.
E-mail address: jqian@nuaa.edu.cn

However, for the performance related issues in complex service systems, there still lacks systematic fault injection studies in literature. Most of the existing fault injection work on web services only briefly discuss how to inject message delays [1, 3, 7, 8, 12, 13, 17, 20] or timeout faults [5, 19, 22] to test the error handling in a system. For composite services, including the ones integrated together via general programming languages like Python and the ones integrated via domain-specific BPEL service orchestration language, how to systematically inject different faults to different positions in order to evaluate the service performance under extreme conditions is still an open question. If the faults are just casually created and injected, the fault revealing ability of fault injection cannot be guaranteed.

To address the above problem, this paper presents a systematic fault injection approach for composite web services toward performance related issues. We firstly define several coverage criteria to guide fault injection, including sub-service single invocation delay coverage, sub-service multiple invocation delay coverage, and sub-service stability coverage. From these coverage criteria, we propose algorithms to derive a fault injection configuration set for a web service. The fault injection process will be conducted automatically under the generated fault injection configurations, and we will use specifications on the service availability, the max response time, the stability of response time, etc. to check whether the performance of the composite service is as expected. The proposed approach can systematically and automatically test performance problems in a composite web service with little dependence on the testers' personal experience.

2. The Overall Approach

The subjects under test of the proposed approach are composite services that are either SOAP-based ones integrated by BPEL language or other ones that are integrated by general programming languages. A typical example of these non BPEL-based web services is the web API of OpenStack [18], which integrates various functionalities via Python to provide a RESTful service interface for cloud management.

For easy explanation, we use BPEL composite services as an example to demonstrate the approach. BPEL is an XML-based language that supports structures like sequence, if, while, invoke, etc. A BPEL service description is just like a C or Java program. It specifies the orders and connections between different sub-services in order to achieve certain high-level functionalities. Figure 1 shows an example BPEL composite service, which invokes sub-services Login, Withdraw, Deposit, etc. to provide an ATM service.

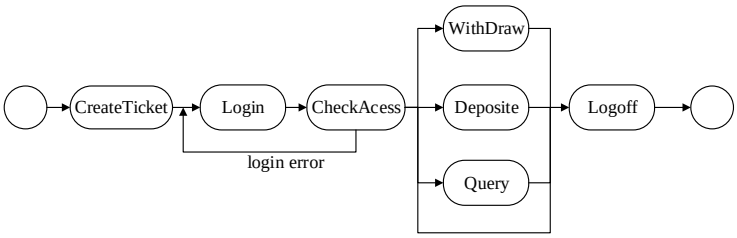

Figure 1. An example ATM composite web service

There are many different testing techniques toward the performance issues in web services, e.g., load testing, stress testing, and so on. The load and stress testing mainly focus on concurrency related issues. As an initial study, however, in this paper, we only focus on the normal performance evaluation problem. More specifically, we independently call a web service for a number of runs under a couple of fault injection configurations and watch the availability of the web service, the service response time, etc. to evaluate its performance. The concurrency related issues will be considered in future work.

A composite service depends on many sub-services to achieve its functionality. The uncertainness occurring on the sub-services may affect the stability of the overall composite service. Before a composite service is online, a service provider usually needs to carefully evaluate its performance, especially under extreme conditions, to make sure that an online service can always meet its service level agreements. Such evaluation can be conducted by largely running the composite service to collect comprehensive performance data, with the hope of luckily covering some extreme conditions. However, the execution time may take too long. In this paper, we use fault injection techniques to trigger the extreme conditions more directly. The situations where a sub-service exhibits boundary performance are injected to help evaluate the worst performance of a composite service. Compared with largely executing composite services, the fault injection approach can speed up the risk-directed performance evaluation process.

The workflow of the proposed approach is shown in Figure 2. In the first step, we analyze the architecture of the composite service under test to get a set of sub-services possibly invoked by it. For BPEL composite services, this can be done by parsing the BPEL description files.

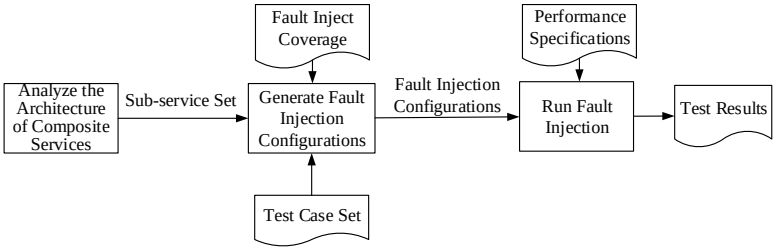

Figure 2. The workflow of the whole fault injection approach

Then, a fault injection configuration set will be derived for the tested composite service according to the sub-service set, a set of test cases for the composite service, and a given fault injection coverage criterion. Each fault injection configuration describes a configuration that is used to run a round of fault injection. The test cases are used to provide service inputs to trigger the execution of the given composite service. We define fault injection coverage criteria to guide comprehensive fault injection. With the defined coverage criteria, different anomalies possibly occurring at different positions of the composite service can be simulated.

Definition 1 (Fault injection configuration, FIC): A fault injection configuration is a tuple $f = \langle t, S, a \rangle$, where t is a test case that feeds inputs for the composite service, S is a sub-service where the fault is injected, and a is an anomaly that will be injected on S.

Definition 1 presents a formal definition of the fault injection configuration. For example, for the ATM composite service shown in Figure 1, assume t_1 is a test case of the composite service, the Login sub-service is called under t_1, and the injected fault is a message delay d on sub-service Login. Then, $<t_1$, Login, $d>$ is a fault injection configuration. It specifies all the options that need to be determined to conduct a round of fault injection for the ATM service.

Finally, the fault injection process will be conducted under these fault injection configurations. The process is demonstrated in Figure 3. We execute fault injection configurations one by one. In each round of fault injection, we feed the test case in a fault injection configuration to the composite service to trigger its execution. When the service execution goes through the sub-service specified in the fault injection configuration, anomalies will be injected. After a round of fault injection, the performance metrics of the composite service will be collected. We then check these metrics against certain performance specifications to determine whether the total performance of the composite service is as expected. The fault injection process can be repeated multiple times so that more performance data can be collected for evaluation.

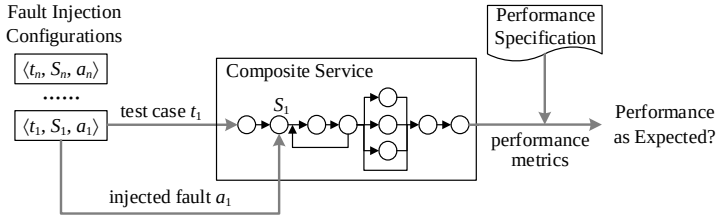

Figure 3. Executing fault injection configurations

3. Performance Specifications

For a web service, either a composite service or an atomic service, a basic requirement on performance is that the service should always give valid responses to its clients, instead of becoming unavailable due to some timeout exception. Besides, there are often requirements on the efficiency of the service. We may want the service's max or average response time to be less than some upper bound. Too slow of a response may affect user experience or cause some high-level operations to fail. The stability of a service's response time is also an important concern. Many service-based systems have implicit assumptions on the response time of web services. If the response time of a service is very unstable, then there will be higher risk for the whole system to exhibit unexpected behavior.

We use specifications to check the requirements on a service's performance attributes. Because a composite service depends on many sub-services to achieve its functionality and the sub-services may even be provided by a third-party, when specifying the performance of a composite service, it is better to not only specify the requirements on the service's performance attributes, but also clearly state under what conditions of the sub-services does the composite service must ensure such performance. The performance specifications used in this work are listed as below.

Availability Specification: For a composite service C composed by a set of sub-services $S(C) = \{S_1, \ldots, S_n\}$, when the response time of each sub-service S_k $(1 \le k \le n)$ is in a range $[L_k, U_k]$, the composite service C must always be available.

Max Response Time Specification: For a composite service C composed by a set of sub-services $S(C) = \{S_1, \ldots, S_n\}$, when the response time of each sub-service S_k $(1 \le k \le n)$ is in a range $[L_k, U_k]$, the max response time of composite service C must be less than an upper bound U_c.

Average Response Time Specification: For a composite service C composed by a set of sub-services $S(C) = \{S_1, \ldots, S_n\}$, when the response time of each sub-service S_k $(1 \le k \le n)$ is in a range $[L_k, U_k]$, the average response time of composite service C must be less than an upper bound $\overline{U}_c$.

Stability Specification: For a composite service C composed by a set of sub-services $S(C) = \{S_1, \ldots, S_n\}$, when the response time of each sub-service S_k $(1 \le k \le n)$ is in a range $[L_k, U_k]$, the standard deviation of composite service C's response time must be less than an upper bound σ_c.

The availability specification is mainly used to check whether a composite service is always available when the performance of its sub-services is still as expected. The max response time specification and the average response time specification check whether the response time of a composite service is too long in the worst situation and in the average situation, respectively. The stability specification is used to check whether the response time of a service is stable.

Our specifications clearly state the relation between the performance of the composite service and the performance of the sub-services. These specifications provide a clear clue for fault injection testing. We can inject boundary performance, e.g., worst response time or worst stability of response time, to the sub-services and watch whether the performance of the composite service violates these specifications. If a violation is found, that indicates the service composition does not carefully consider all the possibilities in the sub-services, and it may result in an unexpected performance.

4. Fault Injection Coverage

Section 3 states the problems that we want to test. With these problems under consideration, several coverage criteria are defined to guide comprehensive fault injection testing of composite web services. Their definitions are as follows.

Definition 2 (Sub-service single invocation delay coverage, SIDC): Let C be the tested composite service with a set of possibly invoked sub-services $S(C)$. If there is a fault injection configuration set $F = \{f_1, f_2, \ldots, f_n\}$ and for each sub-service $S_i \in S(C)$ there exists a fault injection configuration $f_k \in F$ that f_k injects an upper bound response time of S_i on one of S_i's execution, we say the fault injection configuration set F achieves sub-service single invocation delay coverage.

The SIDC coverage can be used to check whether the performance of a composite service is as expected when some of its sub-service exhibit worse response times in one of the sub-service's invocations. Besides performance, with such a coverage, there may also be larger chance to reveal some functional faults if the implementation of the composite service has made bad assumptions on the upper bound of a sub-service's response time, e.g., setting a bad timeout value for the sub-service.

Definition 3 (Sub-service multiple invocation delay coverage, MIDC): Let C be the tested composite service with a set of possibly invoked sub-services $S(C)$. If there is a fault injection configuration set $F = \{f_1, f_2, \ldots, f_n\}$ and for each sub-service $S_i \in S(C)$ there exists a fault injection configuration $f_k \in F$ that f_k injects an upper bound response time of S_i on all of S_i's executions, we say the fault injection configuration set F achieves sub-service multiple invocation delay coverage.

The MIDC coverage can be used to check whether the performance of a composite service is as expected when some of its sub-service exhibit worse response times in all its invocations.

Definition 4 (Sub-service stability coverage, SC): Let C be the tested composite service with a set of possibly invoked sub-services $S(C)$. If there is a fault injection configuration set $F = \{f_1, f_2, \ldots, f_n\}$ that for each sub-service $S_i \in S(C)$, there exists a fault injection configuration set $F' \subseteq F$ satisfying: (1) $|F'|$ is greater than some lower bound K; and (2) the fault injection configurations in F' inject different response time on all the invocations of S_i and the response time values evenly spread in the range of sub-service S_i's possible response time, we say the fault injection configuration set F achieves sub-service stability coverage.

The SC coverage is mainly used to check whether the performance of a composite service is as expected when the response time of its sub-services is unstable. In the definition, the constant number K specifies the minimal number of test executions

that is required to evaluate stability. Doing fault injection following SC coverage can reduce the performance-related risks possibly caused by the uncertainness in the sub-services. This is especially important for some safety-critical service-based systems where we want the response time of a service to be as stable as possible.

5. Generating Fault Injection Configurations

We then automatically generate fault injection configurations toward the coverage criteria defined in Section 4 for fault injection testing.

5.1. Determining the range of response time for sub-services

To generate realistic faults, we need to firstly determine the ranges of possible response time (RPRT) for the sub-services. Without such information, if we just trivially inject a very large constant message delay, the total response time caused by such delay may never happen in the real system, and the fault injection can be meaningless; if we only inject a small message delay, the hidden problems may not be revealed.

The ranges can be determined by two ways. The first way is to determine such ranges according to a tester's experience. However, there is a high risk that such ranges do not match the real performance of sub-services.

Another way to determine the ranges is to firstly execute each sub-service for a number of rounds to collect actual response time data. Then, we can enlarge the range of collected response time to some extent to estimate the possible lower and upper bounds of a sub-service's response time. For example, if the collected range of a sub-service S's response time is $[t_{low}, t_{high}]$ after executing S for 1000 rounds, we can extend that range by 20% to estimate S's range of possible response time, and the final range will be $\text{RPRT}(S) = [80\%*t_{low}, 120\%*t_{high}]$.

The ranges of possible response time determined by the second way can be more reasonable. However, the cost of such determination is sometimes large. One may also consider combining the above two ways to determine the ranges.

5.2. Generating fault injection configurations for SIDC and MIDC

For SIDC and MIDC coverages, we enumerate all the sub-services in a composite service and generate a fault injection configuration for each of them to obtain the final FIC set. The detail of such generation is presented in Algorithm 1. For each FIC, a test case will be selected from the composite service's test case set to trigger the execution of the composite service. The injected faults are message delays on the sub-service's invocations and the delayed time is determined by the range of possible response time of the sub-service (The injected message delays force a sub-service to exhibit its upper bound response time). For SIDC coverage, we intercept a random call of the sub-service to inject a message delay; while for MIDC coverage the message delays are injected to all the calls of the sub-service. The test case, the sub-service where a fault is injected, and the message delay injection option form a fault injection configuration.

The test case selection for SIDC and MIDC coverage is a little different. Assume that the fault is injected on a sub-service S in a fault injection configuration f. For SIDC, we tend to select a test case where S takes a large part of execution time during the execution of the test case for f, so that the impacts of the faults can be maximized. While for MIDC, a test case where sub-service S not only takes a large part of execution time, but also has more invocation times.

Algorithm 1. Generating FICs for SIDC/MIDC coverage
input: C: a composite service;
T: a test case set for C;
output: Fault injection configuration set
begin
  $F := \varnothing$
  foreach sub-service S invoked by composite service C **do**
    select a test case t which covers S from test set T;
    determine the injected response time r according to RPRT(S);
    construct an anomaly description a according to response time r;
    generate fault injection configuration $f = \langle t, S, a \rangle$;
    $F := F \cup \{ f \}$;
  end
  return F;
end

More specifically, let T(*t*, *S*) and N(*t*, *S*) be the time spent on executing a service *S* and the number of invocations to *S* during the execution of a test case *t*, respectively. For a fault injection configuration of a composite service *C* that injects faults to a sub-service *S*, we will select test cases for SIDC and MIDC according to metrics P(*t*, *S*) and M(*t*, *S*):

$$\mathrm{P}(t,S) = \frac{\mathrm{T}(t,S)}{\mathrm{T}(t,C)}, \qquad \mathrm{M}(t,S) = P(t,S) + N(t,S).$$

P(*t*, *S*) is the portion of execution time of sub-service *S* in test case *t* that tests composite service *C*; while M(*t*, *S*) is a mixed metric with both the portion of execution time and invocation times of sub-service *S* under consideration. The test cases with highest P(*t*, *S*) and M(*t*, *S*) metrics will be selected to construct the fault injection configurations for sub-service *S* in composite service *C*.

5.3. Generating fault injection configurations for SC

For SC coverage, a sub-service will correspond to multiple fault injection configurations in the generated FIC set. The detailed algorithm to generate fault injection configurations is presented in Algorithm 2.

For each sub-service *S* in a composite service *C* under test, we firstly select all the test cases covering *S* as candidate bases for fault injection. In Algorithm 2, these test cases form a set *T'*. The test cases in *T'* will then be sorted according to the portion of *S*'s execution time in *C* and the invocation times of sub service *S*, namely according to metric M(*t*, *S*). Then, a fixed *K* number of test cases will evenly be selected from the sorted test case list (*K* is specified in the definition of SC coverage). These selected test cases (*T** in Algorithm 2) form the final fault injection bases. If there are no *K* test cases covering *S* in the given test case set, the existing test cases covering *S* can be duplicated. These base test cases stand for representative situations where sub-service *S* is called in composite service *C*.

Algorithm 2. Generating FICs for SC coverage
input: *C*: a composite service;
T: a test case set for *C*;
K: a threshold number specified by the SC coverage.
output: fault injection configuration set
begin
  F := ∅
  foreach sub-service *S* in composite service *C* **do**
    T' := { test cases covering *S* in test set *T* };
    sort the test cases in *T'* to a list *L* according to metric M(*t*, *S*);
    evenly select *K* test cases from list *L* to get a test set *T**;
    N := the total times of invocations to sub-service *S* in test set *T**;
    D := { evenly select *N* response time values from RPRT(*S*) };
    assign the response time values in *D* to each invocation of *S* in test set *T**;
    foreach test case *t* in *T** **do**
      construct a message delay anomaly *a* according to response time values assigned to the invocations of *S* in *t*;
      generate fault injection configuration *f* = ⟨*t*, *S*, *a*⟩;
      F := *F* ∪ { *f* };
    end
  end
  return *F*;
end

On the base test cases, we firstly count the total number of invocations to sub-service *S*. Let that number be *N*. We then create *N* evenly spread response time values from sub-service *S*'s range of possible response time, i.e., RPRT(*S*). These *N* response time values form a set of unstable respond time with a large standard deviation (set *D* in Algorithm 2). The response time values will be assigned to each invocation of *S* in the base test cases. After that, we will create message delay faults for each base test case according to the assigned response time. If the real response time of a sub-service is less than the assigned response time value, we will add a delay to force the sub-service to respond after reaching the specified response time; otherwise, the assigned response time values are just ignored. A base test case, together with sub-service *S* and the injected message delays on the invocations of *S* under the base test case, form a fault injection configuration. The fault injection configurations on all sub-services form the final FIC set for the tested composite service. The fault injection system will execute the composite services under these fault injection configurations to test the service performance.

6. Experiments

We implemented the proposed fault injection approach and conducted experiments on several BPEL composite services to validate its effectiveness.

The implementation is based on the NUAA SOATest, a service-oriented software testing tool previously designed by our laboratory [25]. SOATest supports generating and executing test cases for SOAP-based web services and RESTful services. It also can inject agents to Java-based service containers to force the network transmissions to go through a proxy. We then intercept SOAP messages on the proxy to inject message delay faults.

In the experiment, we deploy sub-services on Apache Axis2 [23] and deploy BPEL composite services on Apache ODE [24]. ODE provides a library for parsing BPEL files and there is also an extension mechanism in ODE that allows tracing of the sub-services invoked in a composite service by implementing certain monitoring interfaces.

6.1. Experiment settings

The experiment subjects studied in this paper are listed in Table 1. Variants of these subjects are widely used in web service testing research [14, 15]. For all subjects, we assume the performance of the composite services is not as expected, i.e., it may violate certain performance specifications under some situations. The proposed approach will be applied to these subjects to see whether the approach can detect specification violations effectively and efficiently.

Table 1. The experiment subjects

composite service	description	#sub-services
ATM	an ATM service	7
QuickSort	a BPEL implementation of the quick sort algorithm	4
NextDate	a BPEL implementation of the next date problem	3
Triangle	a BPEL implementation of the triangle classification problem	2
MiddleNumber	a BPEL implementation of the middle number determination problem	1

For a composite service without fault injection, a comprehensive approach for performance evaluation (the CE approach for short) is to execute the service for a large number (N_{ce}) of rounds to check the performance attributes. Under such large number of runs, executions leading to bad performance have a high possibility to be covered. In this experiment, N_{ce} is set to 1000, and we assume the performance problems in the experiment subjects can be found under such N_{ce} value.

Executing composite services for a large number of rounds can be very costly. A simplified version of the CE approach (the SE approach) is to run the service for a medium number (N_{se}) of rounds to evaluate the service performance. N_{se} is usually much smaller than N_{ce}. The SE approach is more practical and often used instead of the CE approach in actual testing. It may also detect violations to performance specifications, but with lower possibility.

The approach proposed in this paper (the FI approach) runs the composite service for N_{fi} rounds, where N_{fi} is determined by fault injection configurations generated under a given fault injection coverage criterion. During the N_{fi} rounds of service executions, we inject faults to trigger boundary performance of sub-services more directly, with the hope of speeding up the performance evaluation process.

Table 2. The range of collected response time (ms) for each sub-service

composite service	sub-service	lower bound	upper bound
ATM	CreateTicket	42	133
	Login	78	183
	CheckAcess	54	180
	Query	67	172
	Deposite	118	202
	WithDraw	103	211
	LogOff	42	77
QuickSort	greateThan	4	18
	getNumberFromArray	4	16
	swap	4	12
	assign	4	14
NextDate	bigMonth	3	17
	February	3	14
	Leap	3	19
Triangle	greateThan	4	19
	Add	4	23
MiddleNumber	Copy	5	22

In the experiment, we compare the FI approach and the SE approach to see whether the proposed approach can help

reveal performance problems in composite services. To determine the ranges of possible response times for sub-services, we use the CE approach to collect the ranges of actual response times. The collected ranges are listed in Table 2. (For subject ATM, the response time values are much larger than others because the sub-services in the ATM service operate on a large database.) These ranges will be extended by 20% as the range of possible response times of these sub-services. According to the extended ranges, we inject message delays on sub-service executions to make them exhibit max or unstable response times as specified in the definitions of SIDC, MIDC, and SC.

Because the experiment subjects are assumed to contain performance problems and the CE approach is supposed to have the ability to detect such problems, we write performance specifications for the subjects according to the performance metrics collected under the CE approach. More specifically, we run the service for N_{ce} rounds (1000 rounds) to collect its max response time (MAX), average response time (AVG), and standard deviation of response time (DEV). These data (Table 3) are used as specifications to check the performance attributes of the subject services. An effective performance testing is expected to find the violations of these specifications. Because the service unavailable situations never exist in the experiment, we ignore the results of such specifications in this paper.

Table 3. The performance specifications

composite service	response time		
	upper bound (ms)	upper bound of average (ms)	standard deviation
ATM	715	658	115
QuickSort	1981	1302	394
NextDate	214	171	93
Triangle	325	249	335
MiddleNumber	239	165	154

The experiments try to answer the following research questions.

RQ1: Can the fault injection approach more effectively find violations to performance specifications compared with a normal performance evaluation approach?

We compare the FI approach and the SE approach in how many total times each specification is violated in each subject to answer this research question. The FI approach is expected to effectively find such violations, while the SE approach may occasionally find these performance violations due to luck. The more times the violations are found, the higher the possibility that a testing approach can detect performance problems.

The fault injection will be conducted following three different coverages: SIDC, MIDC, and SC. The corresponding fault injection testing methods are called FI-SIDC, FI-MIDC, FI-SC, respectively. For the availability specification, we check whether an invocation to a composite service throws a timeout exception or returns some special faulty code to see whether the service is unavailable. For the max/average response time specifications, we collect response time data and calculate their average after each composite service invocation to determine whether a specification is violated. For the stability specification, the standard deviation of response time will be calculated after getting every new response time data, and we will check that standard deviation to detect performance specification violations.

To ensure fairness in comparison, we let the SE approach execute the composite services at the same times as that in the fault injection approach. The three testing methods under SE approach corresponding to FI-SIDC, FI-MIDC, and FI-SC are SE-SIDC, SE-MIDC, and SE-SC, respectively.

RQ2: Can the fault injection approach find performance problems more efficiently compared with a normal performance evaluation approach?

We compare the FI approach and the SE approach in how many rounds the composite service are executed in a subject to detect the first specification violation to answer this research question. The fewer the rounds, the better efficiency.

6.2. Experimental results

- Research Question (1)

Table 4 shows the number of composite service invocations under different testing methods and the times of specification violations found by these methods on each subject. Here, for FI-SIDC and FI-MIDC, because the number of fault injection configurations generated for each subject is small (less than 7), to collect more performance data, we repeat the fault injection process on the generated fault injection configurations to make sure a composite service is invoked for at least 50 rounds. The

threshold number *K* in the definition of SC coverage is also tuned to ensure a composite service is invoked for at least 50 rounds in the FI-SC testing. The SE methods invoke the composite services for the same number of rounds as the FI methods. For subjects ATM, NextDate, and MiddleNumber, the composite services only invoke their sub-services for a single time; therefore, the results of FI-SIDC and FI-MIDC are the same, and we do not list the repeated data in Table 4.

Table 4. Times of violations to the performance specifications

composite service	#service invocations			#specification violations						
	SIDC	MIDC	SC	spec.	FI-SIDC	SE-SIDC	FI-MIDC	SE-MIDC	FI-SC	SE-SC
ATM	56	56	56	MAX	15	1	-	-	5	1
				AVG	20	0	-	-	10	0
				DEV	10	2	-	-	22	1
QuickSort	52	52	52	MAX	5	1	13	2	6	0
				AVG	2	0	16	0	10	0
				DEV	9	5	11	8	21	7
NextDate	51	51	51	MAX	13	0	-	-	6	0
				AVG	9	1	-	-	4	1
				DEV	6	0	-	-	11	0
Triangle	50	50	50	MAX	10	0	23	0	14	1
				AVG	7	0	18	1	10	2
				DEV	9	1	11	0	19	0
MiddleNumber	50	50	50	MAX	6	0	-	-	3	1
				AVG	10	0	-	-	5	0
				DEV	8	3	-	-	13	4

Table 5. Rounds of composite service executions where the first specification violation is found

composite service	spec.	FI-SIDC	SE-SIDC	FI-MIDC	SE-MIDC	FI-SC	SE-SC
ATM	MAX	5	29	-	-	11	26
	AVG	3	×	-	-	13	×
	DEV	12	28	-	-	10	37
QuickSort	MAX	14	18	3	14	6	×
	AVG	9	×	2	×	5	×
	DEV	16	19	13	15	6	24
NextDate	MAX	3	×	-	-	8	×
	AVG	4	16	-	-	11	23
	DEV	7	×	-	-	5	×
Triangle	MAX	5	×	1	×	6	31
	AVG	9	×	3	38	9	28
	DEV	15	33	12	×	7	×
MiddleNumber	MAX	11	×	-	-	22	28
	AVG	15	×	-	-	25	×
	DEV	13	23	-	-	5	19

From Table 4, we can see that the FI methods find much more specification violations than the SE methods. The SE methods in many cases may not even find a single time of specification violation. This indicates that with fault injection, performance problems can be more effectively detected compared to performance testing without fault injection. For subjects QuickSort and Triangle, because the FI-MIDC method injects max response time to many sub-service invocations in a test case, they detect even more specification violations than the FI-SIDC method.

- Research Question (2)

Table 5 shows the rounds of composite service executions where the first performance specification violation is found on each subject. From Table 5, we can see that the fault injection methods can find specification violations earlier than the SE methods. The FI methods can constantly detect performance problems in a few rounds, while the SE methods only detect such problems by luck. The fault injection can speed up the performance problem detection process.

In summary, the proposed fault injection approach is both effective and efficient in detecting performance problems for the experiment subjects. Therefore, we consider it a valuable choice for performance testing of composite services.

7. Conclusions

This paper presents a coverage criterion guided fault injection approach to help systematically evaluate the performance of a composite service when the sub-services exhibit boundary behaviors. An initial experimental study shows how the

approach can benefit performance evaluation. The future work includes defining more fault injection coverage criteria to guide more comprehensive testing of various performance related issues, designing better fault injection configuration generation algorithms, and conducting experiments on real service-based systems like OpenStack to further validate the proposed approach.

Acknowledgements

This work is supported by the Science and Technology Planning Project of Jiangsu Province (BY2016003-02) and the China Defense Industrial Technology Development Program under Grant No. JCKY2016206B001 and JCKY2014206C002.

References

1. F. Bessayah, A. Cavalli, W. Maja, E. Martins, and A. W. Valenti, "A Fault Injection Tool for Testing Web Services Composition," In *Proceedings of the 5th International Academic and Industrial Conference on Testing – Practice and Research Techniques*, LNCS, vol. 6303, pp. 137-146, Windsor, UK, September 2010.
2. M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and Verification in Service-Oriented Architecture: A Survey," *Software Testing Verification & Reliability*, vol. 23, no. 4, pp. 261-313, 2013
3. M. G. Fugini, B. Pernici, and F. Ramoni, "Quality Analysis of Composed Services through Fault Injection," *Information Systems Frontiers*, vol. 11, no. 3, page 227, 2009
4. D. Grela, K. Sapiecha, and J. Strug, "A Fault Injection Based Approach to Assessment of Quality of Test Sets for BPEL Processes", In *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 81-93, Angers, France, July 2013
5. S. Harrer, G. Wirtz, F. Nizamic, and A. Lazovik, "Towards a Robustness Evaluation Framework for BPEL Engines," In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*, pp. 199-206, Matsue, Japan, November 2014
6. Z. M. Jiang and A. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Transactions on Software Engineering*, vol. 41, no, 11, pp. 1091-1118, 2015
7. L. Juszczyk and S. Dustdar, "Testbeds for Emulating Dependability Issues of Mobile Web Services," In Proceedings of the 6th World Congress on Services, pp. 683-686 Miami, FL, USA, July 2010
8. L. Juszczyk and S. Dustdar, "Programmable Fault Injection Testbeds for Complex SOA," In Proceedings of the International Conference on Service-Oriented Computing, pp. 411-425, San Francisco, CA, USA, December 2010
9. S. H. Kuk and H. S. Kim, "Robustness Testing Framework for Web Services Composition," *In Proceedings of the IEEE Asia-Pacific Conference on Services Computing*, pp. 319-324, Singapore, Singapore, December 2009
10. L. D. Lago, O. Ferrante, R. Passerone, and A. Ferrari, "Dependability Assessment of SOA-Based CPS with Contracts and Model-Based Fault Injection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 360-369, 2018
11. N. Laranjeiro, M. Vieira, and H. Madeira, "A Robustness Testing Approach for SOAP Web Services," *Journal of Internet Services and Applications*, vol. 3, no. 2, pp. 215-232, 2012.
12. N. Looker and J. Xu, "Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection", In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 163-163, Anacapri, Italy, October 2003
13. D. Manova, I. Manova, S. Ilieva, and D. Petrova-Antonova, "faultInjector: A Tool for Injection of Faults in Synchronous WS," In *Proceedings of the Eastern European Regional Conference on the Engineering of Computer Based Systems*, pp. 99 - 105, Bratislava, Slovakia, Sepetember 2011
14. H. Mei and L. Zhang, "A Framework for Testing Web Services and Its Supporting Tool," In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering* (SOSE), pp. 199-206, Beijing, China, October 2005
15. L. Mei, W. K. Chan, T.H. Tse, B. Jiang, and K. Zhai, "Preemptive Regression Testing of Workflow-Based Web Services," *IEEE Transactions on Services Computing*, vol. 8, no. 5, pp. 740-754, 2015
16. R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Computing Surveys*, vol. 48, no. 3, 2016
17. D. Petrova-Antonova, S. Ilieva, V. Stoyanova, I. Manova, and V. Pavlov, "An Automated Approach for Fault Injection Testing of BPEL Orchestrations," In *Proceedings of the European Conference on Software Process Improvement*, pp. 84-95, 2013.
18. C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, "Failure Diagnosis for Distributed Systems Using Targeted Fault Injection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 503-516, 2017
19. P. Reinecke, A. PA van Moorsel, and K. Woher, "The Fast and the Fair: A Fault-Injection-Driven Comparison of Restart Oracles for Reliable Web Services," In *Proceedings of the International Conference on the Quantitative Evaluation of Systems*, pp. 375-384, Riverside, CA, USA, Sepetember 2006
20. A. Sargeant, P. Townend, J. Xu, and K. Djemame, "Evaluating the Dependability of Dynamic Binding in Web Services," In *Proceedings of the IEEE International Symposium on High-Assurance Systems Engineering*, pp. 139-146, Omaha, NE, USA, October 2012
21. M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing Robustness of Web-Services Infrastructures," In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 131-136, Edinburgh, UK, June 2007
22. Y. Wang, F. Ishikawa, and S. Honiden, "Business Semantics Centric Reliability Testing for Web Services in BPEL," In *Proceedings of the 6th World Congress on Services*, pp. 237-244, Miami, FL, USA, July 2010
23. Apache Axis2, 2018, http://axis.apache.org/axis2/java/core/
24. Apache ODE, 2018, http://ode.apache.org/
25. NUAA SOATest, 2018, http://plase.nuaa.edu.cn/jqian/software/SOATest.html