Towards Dynamic Random Testing for Web Services

Chang-ai Sun[1,3*] , Guan Wang[1], Kai-Yuan Cai[2], Tsong Yueh Chen[4]
[1]School of Computer and Communication Engineering, University of Science and Technology Beijing,Beijing 100083, China
[2]Department of Automatic Control, Beijing University of Aeronautics and Astronautics, Beijing 100191, China
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing 100190, China
[4]Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia
e-mails: casun@ustb.edu.cn, jayjaywg@qq.com, kycai@buaa.edu.cn, tychen@swin.edu.au

***Abstract*—In recent years, Service Oriented Architecture (SOA) has been increasingly adopted to develop applications in the context of Internet. To develop reliable SOA-based applications, an important issue is how to ensure the quality of Web services. In this paper, we propose a dynamic random testing (DRT) technique for Web services which is an improvement of the widely practiced random testing. We examine key issues when adapting DRT to the context of SOA and develop a prototype for such an adaptation. Empirical studies are reported where DRT is used to test two real-life Web services and mutation analysis is employed to measure the effectiveness. The experimental results show that DRT can save up to 24% test cases in terms of detecting the first seeded fault, and up to 21% test cases in terms of detecting all seeded faults, both with the cases of uniformed mutation analysis and distribution-aware mutation analysis, which refer to faults being seeded in an even or clustered way, respectively. The proposed DRT and the prototype provide an effective approach to testing Web Services.**

Keywords- Software Testing; Random Testing; Web Services

I. INTRODUCTION

Service Oriented Architecture (SOA) proposes an application development paradigm [19]. In this context, applications are built upon services that expose functionalities by publishing their interfaces in appropriate repositories, and abstracting entirely from the underlying implementation. Published interfaces may be searched by other services or users and then be invoked. Ensuring the reliability of SOA-based applications becomes crucial particularly when such applications implement important business processes. Testing presents a systematic and practical choice. However, some features of SOA pose challenges for testing Web services [7].

Random Testing (RT) [13] is one of the most widely practiced black-box testing techniques. RT selects test cases from the entire input domain randomly and independently, and it is easy to use; in the meanwhile, RT may be inefficient in some situations because it does not make use of information about software under test (SUT) and the test history. In recent years, many efforts have been made to improve RT [6, 9, 15].

In this paper, we present a dynamic random testing (DRT) for Web services, which is an enhanced version of RT and in the meanwhile an adaptation of DRT to the context of SOA.

*Corresponding author

The contributions of this work include:

- A feasible testing technique for SOA-based applications, including a DRT framework which addresses key issues of testing Web services and a prototype which partially automates the framework,
- Empirical studies which not only validate the feasibility of the proposed DRT framework, but also examine key factors on the effectiveness of DRT, and
- Effectiveness evaluation by means of uniform mutation analysis and distribution-aware mutation analysis.

The remaining of the paper is organized as follows. Section II introduces the underlying concepts of DRT, Web services and mutation analysis. Section III presents a framework DRT for Web services, and a prototype which partially automates the DRT. Section IV describes two case studies where the proposed DRT is used to test two real-life Web services. Section V presents related work and Section VI concludes the paper.

II. BACKGROUND

In this section, we present the underlying concepts of DRT, Web services, and mutation analysis.

A. Dynamic Random Testing (DRT)

It has been observed that faults in programs tend to be clustered [1, 12] and this observation results in the Pareto principle. The principle states that if a test detects one failure in a module, more faults are likely located within the module. Consequently, to be more efficient, the preference should be given to those test cases relevant to the module. However, conventional RT randomly and independently selects tests cases for execution. DRT [6] is proposed to improve testing process of RT by making use of test history data collected in the previous tests.

DRT combines RT and partitioning testing [23] in order to enjoy the benefits of both testing techniques. Assume the test suite *TS* is classified into m partitions denoted as $C_1, C_2, \ldots, C_m$, and each domain C_i has K_i distinct test cases, where $i=1..m$ and $K_1+K_2+\ldots+K_m \geq K$ (K is the number of test cases in *TS*). A DRT algorithm [6] is proposed as follows:

Step 1 Initialize the parameter ε, where $0<\varepsilon<1$.

Step 2 Select a partition C_i according to the given probability distribution $\{p_1, p_2, \ldots, p_n\}$, where p_i is referred to as the probability of C_i being selected, and $\sum_{i=1}^{m} p_i = 1$.

Step 3 Select a test case t from C_i randomly in accordance with the uniform distribution. That is, each distinct

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.26

test case in C_i has an equal probability of $1/ K_i$ being selected.

Step 4 Execute the selected test case t. If a failure is detected by t, increase the probability P_i of test cases in C_i being selected and at the same time decrease the probability P_j of test cases in other partitions as follows:

$$p_j = \begin{cases} p_j - \varepsilon/(m-1); & if\ j \neq i \wedge p_j \geq \varepsilon/(m-1) \\ 0; & if\ j \neq i \wedge p_j < \varepsilon/(m-1) \end{cases},$$ and

$p_i = 1 - \sum_{j \neq i} p_j$, and remove the detected defect from the software under test;

Otherwise, take the following actions:

$$p_i = \begin{cases} p_i - \varepsilon; & if\ p_i \geq \varepsilon \\ 0; & if\ p_i < \varepsilon \end{cases},$$ and

$$p_j = \begin{cases} p_j + \varepsilon/(m-1); & if\ j \neq i \wedge p_i \geq \varepsilon \\ p_j + p_i/(m-1); & if\ j \neq i \wedge p_i < \varepsilon \end{cases}.$$

Step 5 Check whether the test criterion is satisfied. If it is satisfied, stop testing; otherwise, repeat the execution of Steps 2 to 4.

B. Web Services

Web services are a platform-independent, loosely coupled, self-contained programmable web-enabled application that can be described, published, discovered, coordinated and configured using XML artifacts for the purpose of developing distributed interoperable applications [19]. Their descriptions are specified as WSDL files, and their implementations can be written in any programming language. Web services expose their functionalities by publishing interfaces and are deployed in a service container.

C. Mutation Analysis

Mutation analysis [11] is widely used to assess the adequacy of test suites and the effectiveness of testing techniques. It applies some mutation operators to seed various faults into the program under test, and thus generates a set of variants, namely mutants. If a test case causes a mutant to show a behavior different from the program under test, we say that this test case can "kill" the mutant and thus detect the fault injected into the mutant. The mutation score (MS) is normally used to measure how thoroughly a test suite can kill the mutants, which is defined as

$$\text{MS}(p, ts) = \frac{N_k}{N_m - N_e}, \quad (1)$$

where p refers to the program being mutated, ts refers to test suite under evaluation, N_k refers to the number of killed mutants, N_m refers to the total number of mutants, and N_e refers to the number of equivalent mutants. Compared with manually seeded faults, the automatically generated mutants are more similar to the real-life faults, and the MS is a good indicator for the effectiveness of a testing technique [2].

III. APPLICATION OF DRT TO WEB SERVICES

We first propose a framework of DRT for Web services, and then describe a prototype which partially automates the framework.

A. Framework

Considering the principle of DRT and the features of Web services, we propose a framework of DRT for Web services, as illustrated in Figure 1. Components in the left part are relevant to DRT, and Web services under test are located in the top right part. The interactions between DRT components and Web services are depicted in the framework. Test case generation in DRT for Web services still follows the principle of random testing; on the other hand, the selection of next test case is in accordance with the probability distribution. In this way, DRT takes both advantages of random testing and partition testing. We next discuss components in the framework individually.

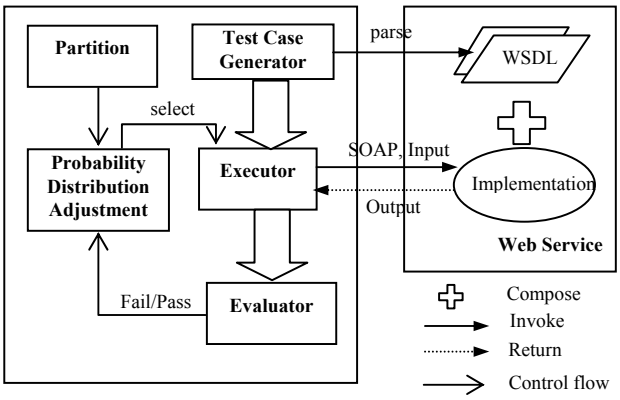

Figure 1. The framework of DRT for Web Services

(1) *Partition*. Partition testing refers to a class of testing techniques [23]. The principles on achieving convenient and effective partitions have been discussed in some literature [5, 8, 23]. We just leverage them in the DRT and detailed discussions are beyond the scope of this paper.

(2) *Test Case Generator*. For each partition, one can randomly and independently generate a test suite with a specified size. This can be done automatically by parsing the WSDL of Web services under test and generating a pool of test data. A WSDL file defines the formats of test cases by the input messages.

(3) *Probability Distribution Adjustment*. The component is responsible for setting the initial probability for each partition. There are two ways to set the initial test profile. One is to make a uniform probability distribution; the other is to set larger probabilities for those core partitions while smaller probabilities for less important ones.

(4) *Executor*: The component receives test suites generated by *Test Case Generator* and selects test cases for execution in accordance with the probability distribution of partitions. During the testing, the component is responsible for converting test cases into input messages, invoking Web services through the SOAP protocol, and intercepting the test results.

(5) *Evaluator*. After one test is completed, the component decides whether the test passes or fails by comparing the actual output with the expected output. With the evaluation result, *Probability Distribution Adjustment* accordingly adjusts the probability distribution.

B. Prototype

Figure 2 shows a snapshot of the prototype which partially automates DRT for Web services. To start, testers need to input the address of Web service being tested, and press "*Parse*" button to analyze input formats and output formats. Next, an operation is selected from the operation list (in the bottom left). As to the partitions and test suites, the prototype provides two options. One is to automatically generate partitions and test suites; the other is to upload the predefined partitions and test suites. If the former is selected, the initial test profile will be automatically set; otherwise, this task is left for testers. Before executing tests (the "*Test*" button), testers are required to set limits on the number of tests ("*Test Repetition Limit*"). During the testing, if a failure is detected without exceeding the limits, the prototype suspends the testing and asks for tester's instruction. Testers can choose to remove defects and continue tests or stop tests. When tests are completed, the test report is summarized in a file which may be used for measuring the effectiveness of DRT.

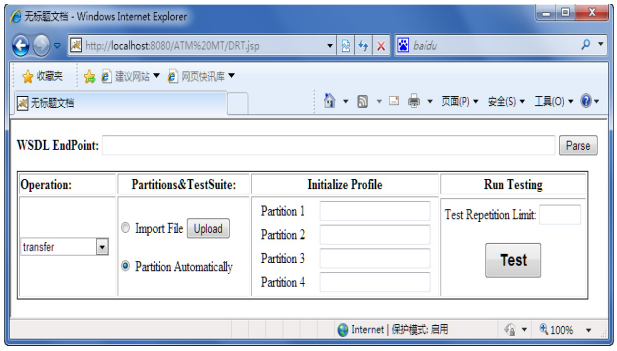

Figure 2. A snapshot of the prototype interface

IV. EMPIRICAL STUDIES

In this section, we describe two case studies where mutation analysis is used to evaluate the effectiveness. The results of the case studies show that DRT outperforms RT and can save around 20% test cases used by RT with respect to both detecting the first defect and all seeded defects.

A. Experiment Settings

To effectively execute experiments, the prototype was employed. Firstly, we employ the *Test Case Generator* to randomly generate test suites for partitions. For the initial test profile, each partition has an equal probability in our experiment. Secondly, we employ the *Executor* to control tests, including selecting test cases, executing tests and intercepting outputs of tests. Thirdly, the *Evaluator* is employed to evaluate the results of tests. Finally, we employ the *Probability Distribution Adjustment* to adjust the probability distribution. Repeat testing until the predefined criterion is satisfied.

In our experiments, we seed faults into the implementation of Web services. This task is automatically done using *MuJava* [17]. Two metrics are defined to measure the effectiveness of RT and DRT. *F-measure* is defined as the number of test cases used to detect the first fault, and *T-measure* is defined as the number of test cases used to detect all faults. Due to the fact that faults in real-life programs are usually clustered [1, 12], mutation analysis should simulate the similarly clustered mutants, in order to deliver reliable measurements. As an illustration, we employ the Pareto Chart to simulate clustered mutants which follows the 80/20 distribution [12]. Such a mutation is called the *Pareto* mutation [22]. We repeat the experiments 50 times to make the evaluation results conclusive, and set probability adjustment parameter ε to 0.02, 0.05, 0.1 and 0.3.

B. Case Study I

An electronic payment system is implemented as Web service and deployed in the Tomcat server [21]. The system offers several features, such as withdrawal, deposit, transfer, query, and each of them is encapsulated as a service port. Among these features, we select the transfer feature for the case study because it is widely practiced in the electronic payment. For the commission fee charging criterion, we refer to Agricultural Bank of China for the calculation rules as shown in Table I. Transfer types I-IV refer to the transfer between two accounts in the same bank and city, in the same bank but different cities, in the same city but different banks, in different cities and different banks, respectively. The implementation consists of 136 lines of Java code.

TABLE I. COMMISSION FEE CALCULATION

	I	II	III	IV
Charge Percentage	0%	0.5%	0.5%	1%
Min(￥)	0	1	1	1
Max(￥)	0	50	50	50
Limit Per Transfer (￥)	50000	50000	50000	50000

By analyzing the WSDL of the transfer feature, we derive the input of the transfer operation, and represent it as a 4-tuple integer vector (A, B, P, M), where

- *A* and *B* denote the sender and recipient account numbers for the transfer transaction, respectively.
- *P* denotes the transfer type. Its value ranges from 0 to 3, corresponding to type I to IV in Table I.
- *M* denotes the amount of a transfer transaction.

According to the specification of the transfer feature in Table I, we have two partition schemas shown in Table II. Here, *rate (p)* denotes commission ratio of *p* in Table I. Scheme I only considers partition on the transfer type *P*, while Scheme II considers partitions on both the transfer type *P* and the transfer amount *M*.

In our experiments, a total of 139 mutants are derived by *MuJava*. Among them, 10 mutants are equivalent ones and thus are excluded from experiments. We get a family of 71 clustered mutants from evenly distributed 129 mutants. As a result, 58 mutants are discarded.

Table III summarizes the effectiveness of RT and DRT in term of *F-Measure* with uniform mutation, varying partition schema and probability adjustment parameters. The *NF* column represents the average values of *F-Measure*. The

TABLE II. TWO PARTITION SCHEMA

	Scheme I	Scheme II
A	N/A	N/A
B	N/A	N/A
P	0	0
	1	1
	2	2
	3	3
M	N/A	M<=0
		0<M≤1/rate(P)
		1/rate(P)<M≤50/rate(P)
		50/rate(P)<M≤50000
		M>50000

"*Improvement (%)*" column represents effectiveness improvement of DRT over RT, which is calculated by the following formula: $(N_{F\text{-}RT}\text{-}N_{F\text{-}DRT})\ /N_{F\text{-}RT}$ x100%, where $N_{F\text{-}DRT}$ and $N_{F\text{-}RT}$ represents the average *F-Measure* value of DRT and RT, respectively. The "*SD*" column represents the standard deviation over 50 trials. Since there is only one fault within each mutant, both *Uniform* mutation and *Pareto* mutation deliver the same result when the *F-measure* is employed. We observe from Table III that

- DRT runs fewer test cases to detect the first failure than RT. This means that DRT can detect the first failure faster than RT. DRT considerably outperforms RT by 15.51% to 23.95% in terms of *F-Measure*.
- Partition schema has a small impact on the effectiveness of DRT. For the same ε, DRT has very similar effectiveness with two partition schema. We further investigate the reason behind this, and discover that: (1) faults are evenly seeded into partitions in the context of Uniform mutation; (2) DRT selects test cases among these partitions in an exploratory way before detecting the first failure; (3) the exploratory process is not severely affected by the number of partitions.
- The settings of probability adjustment parameter ε have a significant impact on the effectiveness of DRT.

Table IV summarizes the effectiveness of RT and DRT in term of *T-Measure* with the *Pareto* mutation, varying partition schema and probability adjustment parameters. The *NT* column represents the average values of *T-Measure*. The "*Improvement (%)*" column represents performance improvement of DRT over RT, which is calculated by the following formula: $(N_{T\text{-}RT}\text{-}N_{T\text{-}DRT})\ /N_{T\text{-}RT}$ x100%, where $N_{T\text{-}DRT}$ and $N_{T\text{-}RT}$ represents the average *T-Measure* value of DRT and RT, respectively. We observe from Table IV that

- DRT uses fewer test cases to detect all seeded faults than RT. On one hand, when DRT detects a fault in some partitions, the probability of selecting test cases in those partitions will be increased; on the other hand, the seeded faults are clustered in some partitions in the context of *Pareto* mutation. In this context, DRT detects more faults in a greedy way in those partitions with clustered faults and thus can detect all faults with fewer test cases. In general, DRT outperforms RT by up to 20.10% in terms of *T-Measure*. This means that DRT can save more efforts to improve the quality of software than RT.
- Partition schema have a significant impact on the effectiveness of DRT. With partition scheme II, DRT clearly outperforms RT by 9.2% to 20.10%; with partition scheme I, the improvement of DRT over RT is marginal. This can be explained as follows. Partition scheme I only makes partitions on the transfer type *P* while not on the transfer account *M*, while the latter has a heavy influence on program under test, so partition scheme I is not a really complete partition scheme which has not taken the full advantage of greedy and exploratory characteristic of DRT. On the contrary, partition scheme II derives more partitions than partition scheme I, and DRT achieves a better effectiveness than RT with partition scheme II.
- The settings of probability adjustment parameter ε show an inconsistent impact on the effectiveness of DRT. For partition scheme I, the impact is fluctuating, while for scheme II, the effectiveness significantly improves with the increase of ε.

C. CASE STUDY II

In a supply chain management application for a consumer electronics retailer [20], consumers send requests to the retailer and order products. The system is implemented by orchestrating several Web services. Among them, *Warehouse Service* is selected as subject program because it is the most important part of the system. By analyzing its WSDL file, we derive that its input is a collection of entries consisting of *productID* and *quality*. To test such a Web service, we divide the input domain into partitions according to *number of different products*, *product ID*, and *quantity*, as shown in Table V.

Table III Evaluation Results of DRT and RT using F-Measure and Uniform Mutation

		Partition Scheme I			Partition Scheme II		
		N_F	Improvement (%)	*SD*	N_F	Improvement (%)	*SD*
RT		8.77	N/A	0.57	8.77	N/A	0.57
DRT	ε=0.02	7.4	15.62	0.29	7.41	15.51	0.42
	ε=0.05	6.99	20.30	0.32	7.08	19.27	0.38
	ε=0.1	6.81	22.35	0.29	6.72	23.38	0.38
	ε=0.3	6.82	22.23	0.38	6.67	23.95	0.29

Table IV Evaluation Results of DRT and RT using T-Measure and Pareto Mutation

		Partition Scheme I			Partition Scheme II		
		N_T	Improvement (%)	*SD*	N_T	Improvement (%)	*SD*
RT		161.4	N/A	5.01	161.4	N/A	5.01
DRT	ε=0.02	165	-2.23	6.92	146.52	9.22	6.85
	ε=0.05	160.38	0.63	8.98	138.12	14.42	7.29
	ε=0.1	157.04	2.70	7.24	134.84	16.46	9.73
	ε=0.3	158.64	1.71	8.06	128.96	20.10	10.24

TABLE V. PARTITION SCHEME

Input Vectors	**Partitions**
Types of products	one
	More than one
Product ID	valid
	invalid
quantity	valid
	invalid

In our experiment, a total of 208 mutants are derived by *MuJava*, and 45 of them are equivalent ones. We get a family of 41 clustered mutants from 163 evenly distributed mutants. As a result, 122 mutants are discarded.

Tables VI and VII summarize the effectiveness of RT and DRT in term of *F-Measure* with uniform mutation and *T-Measure* with Pareto mutation, respectively. From Tables VI and VII, we have the following observations:

- DRT uses fewer test cases to detect the first failure than RT. DRT outperforms RT by 4.52% to 8.20% in terms of *F-Measure*.
- DRT uses fewer test cases to detect all seeded faults than RT. In general, DRT outperforms RT by up to 15.21% in terms of *T-Measure*.
- The settings of probability adjustment parameter ε have a significant impact on the performance of DRT.

TABLE VI F-MEASURE RESULT OF WAREHOUSE SERVICE WITH UNIFORM MUTATION

		N_F	Improvement (%)	*SD*
RT		11.31	0.00%	0.87
DRT	ε=0.02	9.96	+8.20%	0.44
	ε=0.05	10.57	+6.54%	0.36
	ε=0.1	10.49	+7.25%	0.38
	ε=0.3	10.36	+4.52%	0.48

D. Discussions

Through the above case studies, we observe that the probability adjustment parameter ε demonstrates a significant impact on the performance of DRT. That is, the probability adjustment parameter plays a key role in the performance of DRT and should be carefully designed. In general, the setting of probability adjustment parameter ε has a close relation with the number of partitions. With the same partition scheme, the larger ε, the more acute the updates of test profile are; the smaller ε, the more stable the updates of test profile are. An increase of the number of partitions may weaken the updates of test file associated with a larger ε. Thus, one should set appropriate ε in terms of the ratio of ε over the number of partitions.

TABLE VII T-MEASURE RESULT OF WAREHOUSE SERVICE WITH THE PARETO MUTATION

		N_T	Improvement (%)	*SD*
RT		184.00	0.00%	5.42
DRT	ε=0.02	163.67	+11.05%	6.19
	ε=0.05	156.02	+15.21%	9.99
	ε=0.1	160.32	+12.87%	9.44
	ε=0.3	164.58	+10.55%	8.08

V. RELATED WORK

We describe several related work on improvements on RT and testing techniques for Web services.

A. Improvements on RT

Many efforts have been made to improve RT. Chen et al. [9] proposed adaptive random testing (ART) to achieve more even distribution of test cases generated by RT. Various ART algorithms have been developed and their effectiveness are validated by simulation experiments [9, 10]. Some strategies were proposed to improve the fault detection effectiveness of RT based on an analysis of the distribution of test cases generated by RT and the phenomenon that ART deteriorates in the high dimensional input domains [15]. Cai et al [5] proposed adaptive testing (AT) which dynamically adjusts tests in partitions by making use of feedback information such as test data collected online. DRT [6] was proposed to random-partition testing process by combining dynamic partitioning technique and AT.

B. Testing techniques for Web services

In recent years, various testing techniques for Web services have been proposed. Bartolini et al. [4] developed a tool called TAXI that generates test cases for Web services based on WSDL specifications. Bai et al. [3] proposed an ontology-based partition testing approach for Web services. Lenz et al. [16] applied model-driven approaches into the testing of Web services. Sun et al. [21] proposed a metamorphic testing technique to address the outstanding oracle problem with testing Web services. Many other testing methods for Web services can be found in the literature, such as contract-based

Web services testing [14], fault-based Web services testing [18], etc.

Compared with existing approaches, the proposed DRT for Web services has the following advantages: (1) it is easier to use. DRT is applicable whenever partition testing is applicable; (2) it is more efficient because it enhances partition testing with dynamic test profiles updates and the resulting overhead is very limited.

VI. CONCLUSIONS

We have presented a dynamic random testing technique for Web services to address the challenges of testing SOA-based applications. The proposed technique employs random testing to generate test cases, and selects test cases for execution from different partitions in accordance with the test profiles of partitions which are dynamically updated in response to the test date collected online. In this way, the proposed test technique takes benefits of both random testing and partition testing, and provides an effective and efficient approach to testing Web services.

A framework was proposed to examine key issues when applying DRT to testing Web services. To make our approach practical and effective, a prototype was further developed. To validate the feasibility and effectiveness of our approach, two case studies were conducted where two real-life Web services were used as subject program and mutation analysis was employed for measuring effectiveness. The results of case studies show that compared with random testing, DRT can save up to 24% test cases in terms of detecting the first seeded fault, and up to 21% test cases in terms of detecting all seeded faults. This means that DRT can detect the first fault or all faults using fewer test cases than RT.

In our future work, we plan to conduct experiments on more Web services to further validate the effectiveness and identify the limitations of our approach.

VII. ACKNOWLEDGMENTS

Authors thank to Zi-qiang Zhang from University of Science and Technology Beijing for his help to implement the prototype presented in the paper. This research is supported by the National Natural Science Foundation of China (Grant Nos. 60903003, 60973006), the Beijing Natural Science Foundation of China (Grant Nos. 4112037, 4112033), the Fundamental Research Funds for the Central Universities (Novel Testing Techniques and Tool for SOA), the Open Funds of the State Key Laboratory of Computer Science of Chinese Academy of Science (Grant No. SYSKF1105), and a discovery grant of the Australian Research Council (Grant No.DP0771733).

REFERENCES

[1] P. E. Ammann, and K. C. Knight. "Data Diversity: An Approach to Software Fault Tolerance". *IEEE Transactions on Computers,* 1988, 37(4):418~425.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In *Proceedings of the 27th International Conference on Software Engineering* (ICSE 2005). ACM Press, 2005, pp402-411.

[3] X. Bai, S. Lee, W.-T. Tsai, and Y. Chen. "Ontology-based Test Modeling and Partitioning Testing of Web Services", In *Proceedings of the 6th International Conference on Web Services* (ICWS 2008), IEEE Computer Society, 2008, pp465-472.

[4] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. "WS-TAXI: A WSDL-based Testing Tool of Web Services", In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation* (ICST'09), IEEE Computer Society, 2009, pp326-335.

[5] K. Y. Cai, T. Jing and C. G. Bai. "Partition Testing with Dynamic Partitioning", In *Proceedings of the 29th Annual International Computer Software and Application Conference* (COMPSAC 2005), IEEE Computer Society, 2005, pp113-116.

[6] K. Y. Cai, H. Hu, C. Jiang and F. Ye. "Random Testing with Dynamically Updated Test Profile", In *Proceedings of the 20th International Symposium on Software Reliability Engineering* (ISSRE 2009), Fast Abstract 198.

[7] G. Canfora and M. Penta. "Service Oriented Architecture Testing: A Survey", *LNCS 5413*, Springer-Verlag, Heidelberg, 2009:78–105.

[8] T. Y. Chen and Y. T. Yu. "On the Relationship Between Partition and Random Testing", *IEEE Transactions on Software Engineering*, 1994, 20(12): 977-980.

[9] T. Y. Chen, F. -C. Kuo, R. Merkel and T.H. Tse. "Adaptive Random Testing: The ART of Test Case Diversity". *Journal of Systems and Software*, 2010, 83(1): 60-66.

[10] T. Y. Chen, F. -C. Kuo and C. Sun. "Impact of the Compactness of Failure Regions on the Performance of Adaptive Random Testing". *Journal of Software*, 2006, 17(12): 2438-2449.

[11] R. A. DeMillo, R. J. Lipton and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, 1978, 11(4): 31-41.

[12] N.E. Fenton and N. Ohlsson. "Quantitative analysis of faults and failures in a complex software system", *IEEE Transactions on Software Engineering*, 2000, 26(8):797-814.

[13] R. Hamlet. "Random Testing". *Encyclopedia of Software Engineering*. John Wiley & Sons, New York, NY. 2002.

[14] R. Heckel and M. Lohmann. "Towards Contract-based Testing of Web Services", *Electronic Notes in Theoretical Computer Science.* 2005, 116 :145-156.

[15] F.-C. Kuo, K.Y. Sim, C. Sun, S.F. Tang and Z.Q. Zhou. "Enhanced Random Testing for Programs with High Dimensional Input Domains", In *Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering* (SEKE 2007), 2007, pp135-140.

[16] C. Lenz, J. Chimiak-Opoka and R. Breu. "Model Driven Testing of SOA-based Software", In *Proceedings of the Workshop on Software Engineering Methods for Service-Oriented Architecture* (SEMSOA 2007), 2007, pp99-110.

[17] J. Offutt, Y.S. Ma and Y.R. Kwon. "An Experimental Mutation System for Java", *ACM SIGSOFT Software Engineering Notes*, 2004, 29(5): 1-4.

[18] J. Offutt and W. Xu. "Generating Test Cases for Web Services using Data Perturbation", *ACM SIGSOFT Software Engineering Notes*, 2004, 29 (5):1-10.

[19] M. Papazoglou, P. Traverso, S. Dustdar and F. Leymann. "Service-Oriented Computing: A Research Roadmap", *International Journal on Cooperative Information Systems (IJCIS)*, 2008, 17(2):223-255.

[20] R. Rekasius. "Preview of the WS-I sample application", http://www.ibm.com/developerworks/webservices/library. 2011.

[21] C. Sun, G. Wang, B. Mu, H. Liu, Z.S. Wang and T.Y. Chen. "A Metamorphic Relation-Based Approach to Testing Web Services Without Oracles", *International Journal on Web Service Research (JWSR)*, 2012, 9(1): 51-73.

[22] C. Sun, G. Wang, K.Y. Cai and T.Y. Chen. "Towards Distribution-aware Mutation Analysis", submitted for publication, 2012.

[23] E.J. Weyuker and B. Jeng. "Analyzing Partition Testing Strategies", *IEEE Transactions on Software Engineering*, 1991, 17(7):703-711.