Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications *

W. K. Chan †, ‡ S. C. Cheung§ and Karl R. P. H. Leung¶

Technical Report HKUST–CS05–11
20 July 2005

HKUST
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Kowloon, Hong Kong

{wkchan, sccheung}@cs.ust.hk and kleung@computer.org

Abstract

Testing applications in service-oriented architecture (SOA) environments needs to deal with issues like the unknown communication partners until the service discovery, the imprecise black-box information of software components, and the potential existence of non-identical implementations of the same service. In this paper, we exploit the benefits of the SOA environments and metamorphic testing (MT) to alleviate the issues.

We propose an MT-oriented testing methodology in this paper. It formulates *metamorphic services* to encapsulate services as well as the implementations of metamorphic relations. Test cases for the unit test phase is proposed to generate follow-up test cases for the integration test phase. The metamorphic services will invoke relevant services to execute test cases and use their metamorphic relations to detect failures. It has potentials to shift the testing effort from the construction of the integration test sets to the development of metamorphic relations.

Keywords: service-oriented architecture, unit testing, integration testing, metamorphic testing infrastructure.

* This research is supported by grants of the Research Grants Council of Hong Kong (Project Nos. HKUST 6170/03E and CITYU 1195/03E).

† All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, The University of Hong Kong of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Tel: (+852) 2358 7016. Fax: (+852) 2358 1477. Email: wkchan@cs.ust.hk. (Part of the research was performed when Chan was with The University of Hong Kong, Pokfulam, Hong Kong.)

‡ Hong Kong University of Science and Technology Clear Water Bay, Hong Kong

§ Hong Kong University of Science and Technology Clear Water Bay, Hong Kong

¶ Hong Kong Institute of Vocational Education, Tsing Yi, Hong Kong

1 Introduction

Service-Oriented Architecture (SOA) is a model for a kind of distribution applications such as the Web services [15]. As SOA applications become popular, assuring their quality is important. This paper proposes a methodology for software testing, as software testing is a major mean to assure the correctness of software.

A basic element of SOA is a *service* which is a software module, capable to (*i*) describe itself, (*ii*) be discovered, and hence located, by peer services, and (*iii*) communicate with peer services using machine-readable messages with well-defined interfaces. Since services are discovered dynamically, there is no need to codify, for example, the relationship between two particular services in the source codes explicitly. Services are thus loosely integrated together. This configuration setting greatly enhances their potentials to construct complicated SOA applications.

Nonetheless, as services are loosely coupled, an implementation to offer a service to other services becomes difficult to be predetermined. Moreover, as services are usually developed on heterogeneous platforms [7] and by different parties, knowing all their implementation details is impractical in real-life practice. Hence, although SOA applications can be considered as distributed applications, white-box software testing techniques are less attractive than their black-box counterparts.

Unfortunately, there are at least two testing problems from the black-box perspectives. First, different implementations of a service may behave differently. Test results of a particular implementation cannot reliably treat as the expected ones of other implementations of the same service. Second, the expected behavior of a service seldom exists precisely. Both of them pose testing challenges to check the correctness of such applications.

The main contributions of this paper are as follows.

1. It exploits the *benefits* of dynamic discovery of services to support service testing. Based on this observation, it proposes a novel notion of *metamorphic service*, a kind of access wrapper that encapsulates services under test and supports a metamorphic testing approach.

2. It proposes a testing methodology atop metamorphic services. It allows cross-validation of test results under the SOA framework. Hence, it alleviates the test oracle problem for SOA applications.

3. It supports testers to concentrate on generating test sets for unit test, and delegate the automatic construction (and the automatic checking of the results) of follow-up test cases for both unit test and integration test to the metamorphic services. The infrastructure has potentials to shift the testing efforts from the construction of integration test sets to the development of metamorphic relations.

The rest of the paper is organized as follows: Section 2 introduces the preliminaries of our testing methodology proposal. The proposal and an illustrative example will be presented in Section 3 and Section 4. Next, Section 5 will compare our work and related work. Finally, discussions, conclusions and future work are summarized in Section 6.

2 Preliminaries

In this section, we first revisit metamorphic testing. Next, we describe the assumptions and some useful terminologies relevant to our work.

2.1 Metamorphic relation (MR)

A metamorphic relation is an existing or expected relation over a set of distinct inputs and their corresponding outputs for multiple executions of the target function [5, 6]. It can be formally described as follows: Suppose that f is an implemented function under test. Given a relation r over n distinct inputs, $x_1, x_2, \ldots, x_n$, the corresponding n computation outputs $f(x_1), f(x_2), \ldots, f(x_n)$ must induce a necessary property r_f. A metamorphic relation MR_f of f over n inputs and n outputs can be defined as follows:

$$\begin{aligned} MR_f = \{&(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \\ &| r(x_1, x_2, \ldots, x_n) \Rightarrow \\ &\quad r_f(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))\} \end{aligned}$$

2.2 Metamorphic testing (MT)

Metamorphic testing (MT) [4, 6] uses successful test cases to alleviate the test oracle problem, a problem that a test oracle is unavailable or expensive to be used. Given a program P of target function f with input domain D. A set of test cases $T (= \{t_1, \ldots, t_k\} \subset D)$ can be selected according to any test case selection strategy. Executing the program P on T produces outputs $P(t_1), \ldots, P(t_k)$. When they reveal any failure, testing stops and debugging begins. On the other hand, when no failure is revealed, the metamorphic testing approach can be applied to continue to verify whenever some necessary property of the target function f is satisfied by the implementation P. MT constructs *follow-up* test set T' $(= \{t'_1, \ldots, t'_m\} \subset D)$ automatically from the initial successful test set T, with the reference to some given metamorphic relation.

Let us consider an example. Suppose that S be an expected USD/HKD exchange service, accepting deal orders in USD and returning deal orders in HKD; x and y be two deal orders; $g(\,)$ be a function that accepts a deal order and returns its deal order amount. Further suppose that the following metamorphic relation is given:

$$\begin{aligned} &MR_a(x, y, S(x), S(y)) : \\ &\quad 2g(S(y)) = g(S(x)) \text{ if } g(x) = 2g(y) \end{aligned}$$

Consider a test case x_1 = "a deal order of US$20 000". MT automatically constructs y_1 = "a deal order of US$10 000" based on the condition $g(x) = 2g(y)$ of MR_a. Suppose P is an implementation of S. If the comparison $2g(P(y_1)) = g(P(x_1))$ fails, the MT approach reveals a failure due to the pair of failure-causing inputs x_1 and y_1. In general, x_1 is termed as the *original test case*, and y_1 is termed as the *follow-up test case*. We refer readers to [3–6, 14] for more details about metamorphic testing.

2.3 Assumptions and Some Terminologies

We make the following assumptions: First, in SOA applications, it is viable to add metamorphic services (see Section 3.1), a service encapsulating a service under test to send and receive messages, and to be discovered by other services. It is based on the understanding that SOA services are loosely coupled amongst themselves. Second, we treat a *message* in a SOA application as a *test case* or a *test result* of a service [11, 12]. Third, any service is capable to process a sequence of functions defined in a message. Finally, the program source codes of the metamorphic services are available. As program instrumentation [1] is a common kind of techniques to instrument codes into a program, we further assume that it is possible to add metamorphic relations to metamorphic services. Have said that, the instrumentation procedures for metamorphic relations need further investigation.

We use the following terminologies: A *service* is a software module having the three characteristics defined in Section 1; *unit test* carries its usual meaning as a method of testing the correctness of a particular module; *integration test* refers to the method of testing the correctness of a combined set of individual modules.

3 A Service Testing Methodology

In this section, a service testing methodology will be introduced. In particular, we describe how metamorphic services (*MS*) help testers to conduct the integration testing, based on the activities in the unit test phase.

3.1 Metamorphic Service (*MS*)

We define a metamorphic service as a service which is also *access wrapper* [9, 16], imitating the data and application access paths of a service, the service encapsulated by the metamorphic service. Apart from forwarding messages on behalf of its encapsulated service, a metamorphic service also embodies the program logics (that is, the implementation of metamorphic relations) to compute follow-up message(s) from an incoming (or outgoing) message. Since it is a service, it discovers applicable service implementation(s) to receive a follow-up message. It also forwards the follow-up message to the latter service to process. All relevant test results are then received by an *MS*. This *MS* verifies the test results by using its metamorphic relations. Any violation of any relevant metamorphic relation indicates a failure.

3.2 The Methodological Steps

The major steps of the methodology to support both the unit test and the integration test are as follows:

Unit Test Phase:

(UT-1) Construct a metamorphic service (*MS*) for each service under test (*S*) for unit test.

- (*a*) For each *MS*, design metamorphic relations, say MR_u, applicable to test *S*.
- (*b*) Implement MR_u in the corresponding *MS*.

(UT-2) Construct a test set *TS* for service *S*.

(UT-3) *MS* relays every (original) test case t_o in *TS* to *S*.

(UT-4) For each t_o, *MS* uses its MR_u to compute the follow-up test cases, say t_f.

(UT-5) *MS* forwards every t_f of t_o to *S* to execute.

(UT-6) *MS* uses its MR_u to detect failure.

(UT-7) *MS* reports any detected failure.

Integration Test Phase:

(IT-1) For each *S*, find its set of interacting services, Φ_S.

(IT-2) For each service ϕ_S in Φ_S, design the metamorphic relations MR_i applicable to test the interactions between S and ϕ_S.

(IT-3) Generate the follow-up test cases from TS constructed in Step (UT-2).

(a) For each t_f or t_o applicable to an MS, if it triggers an outgoing message m_o, then MS uses its MR_i to compute follow-up messages, say m_f, based on m_o.

(b) For each t_f or t_o applicable to a MS, and for each MR_i applicable, MS uses the MR_i to compute their follow-up messages, say m_f.

(IT-4) For each m_f, MS uses its MR_i to compute a test result collection message m_{tr}.

(IT-5) MS discovers a service T to process both m_f and m_{tr}. It sends T the composite message (m_f, m_{tr}).

(IT-6) MS receives the test results from T, and applies MR_i to detect failure.

(IT-7) MS reports any detected failure.

We would like to note the following. First, both steps (UT-1a) and (IT-2) require testers to design metamorphic relations. The design of such relations is not within the scope of this paper, however. We refer interested readers to [4] for the case study on the selection of effective metamorphic relations for more details.

Second, step (UT-2) constructs test sets for the unit testing purpose. There are many black-box test case selection techniques [1] available. Testers can apply their criteria to select appropriate techniques to use. Steps (UT-4)–(UT-7), and (IT-3)–(IT-6) are the standard applications of metamorphic testing. In particular, step (IT-5) suggests using the SOA environments to discover a target service. Also, it constructs a composite message, which is forwarded to the target service to process the follow-up message. It also informs the target service to forward the test results to an intended metamorphic service.

Third, a follow-up test case can be an outgoing or incoming message of a service. Steps (IT-3a) and (IT-3b) represent these two choices, respectively.

Last, but not the least, the testing property of a MS can be designed to turn on or off, or be loaded with new definitions of metamorphic relations at run time. Hence, the methodology has potentials to support *on-line testing*.

In the rest of Section 3, we continue to describe its support to the unit test and the integration test. This is followed by an illustrative example based on a foreign exchange dealing system in Section 4.

3.3 The Support To The Unit Test Phase

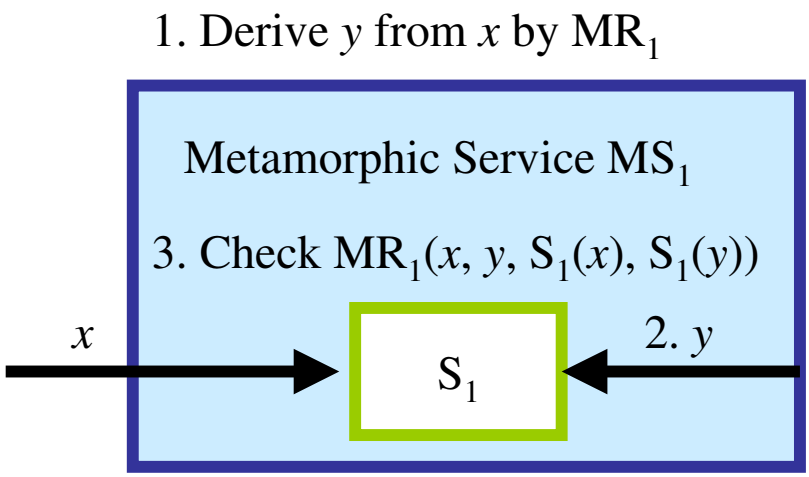

Figure 1: A unit testing scenario

Figure 1 depicts a scenario for the unit test phase. A service S_1 is encapsulated by a metamorphic service MS_1 having a metamorphic relation MR_1.

It shows steps (UT-4)–(UT-6) of the methodology. MS_1 first computes a follow-up message y based on message x. The message is then executed by service S_1. The outputs of service S_1 for both inputs x and y are compared using MR_1.

3.4 The Support To The Integration Test Phase

Figure 2 depicts a scenario for the integration test phase. It supports the checking of test results across services. The service S_1 will produce an output $S_1(x)$ for input x. This output will be forwarded to the service S_4. The metamorphic service MS_1 of S_1 will compute a follow-up message $S_1(x)$' from $S_1(x)$, based on a metamorphic relation MR_2. From the service registry, it discovers the service S_3 to process this follow-up message. Hence, it forwards $S_1(x)$' to S_3. In this particular scenario, both S_3 and S_4 forward their respective results to MS_1. MS_1 then uses MR_2 to detect any failure from the test results.

Figure 3 shows another integration testing scenario. The mechanism is similar to that of Figure 2, except that MS_1 generates follow-up test cases based on an input, instead of an output, of S_1.

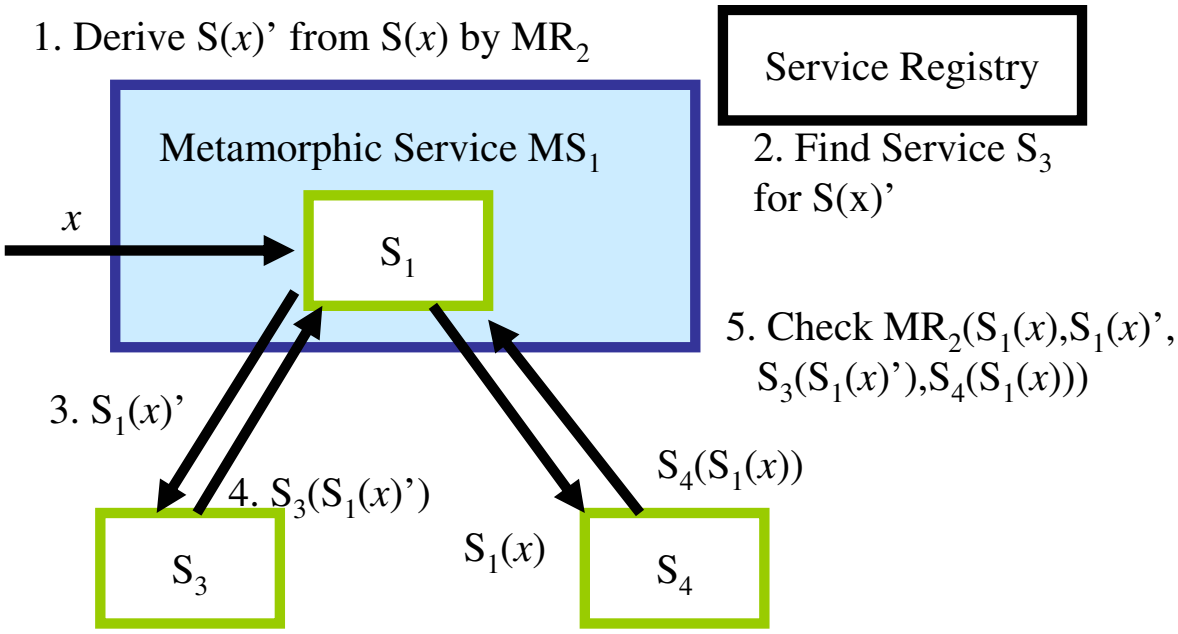

Figure 2: Integration testing scenario one

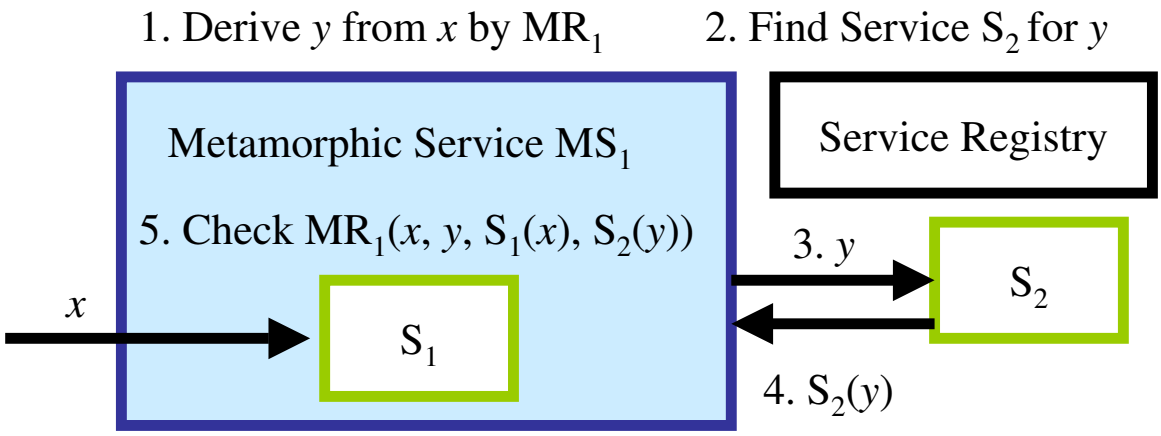

Figure 3: Integration testing scenario two

3.5 The Differences Between The Two Integration Testing Scenarios

MR_2 in Figure 2 could also be a metamorphic relation for a metamorphic service, say MS_4, of the service S_4. In this case, when MS_4 receives the message $S_1(x)$, it may also generate the message $S_1(x)$', based on MR_2. Since the registry is public, MS_4 may also be able to discover S_3. MS_4 then forwards the message $S_1(x)$' to S_3 accordingly. The latter may also be able to forward the test results to MS_4. In other word, the output checking scenario illustrated in Figure 2 seems to be symmetric to the input checking scenario illustrated in Figure 3.

However, MS_1 in Figure 2 may utilize the knowledge related to x to constrain its metamorphic relations such as MR_2. In general, this kind of information is unavailable to neither S_4, nor MS_4. Also, owing to this kind of extra knowledge, MS_1 may discover S_3, but not MS_4. Likewise, an incorrect relationship between the actual test runs of an original test case and its follow-up test cases for MS_1 may not violate relevant metamorphic relations of MS_4. On the other hand, without those special constraints, MS_4 may generate other follow-up test cases to expose failures, which cannot be detected by MS_1. Hence, in general, the former scenario cannot replace the latter, and vice versa.

Having said that, we note that the target functions of S_4 and S_1 may differ. Intuitively, when a function is provided by S_1, S_1 needs not to invoke the service of S_4 to access that function indirectly and unreliably. It simply invokes its own function instead. Hence, although it is interesting theoretically to compare the relative fault detecting capabilities of a metamorphic relation when using these two approaches, the chance of encountering such opportunities is, intuitively, low.

4 A Sample Scenario

In this section, our methodology will be further illustrated by an example. We first describe an application and its faults. Then, we illustrate how these faults can be revealed at the unit test phase and the integration test phase using our testing methodology. We choose to use test cases of the unit test phase as the original test cases of the integration test phase in the illustration.

4.1 The Problem Description

Figure 4 shows a foreign exchange dealing service application with five services, namely FXDS1 to FXDS5. In particular, FXDS2 is embraced by a metamorphic service, referred as *MS*. It has three metamorphic relations, namely MR1 to MR3. Let us restrict our discussion to the exchange of US dollars to Renminbi (that is, RMB, the Chinese currency). A bank normally offers cross-currency rates. A sample USD/RMB exchange rate is a pair of values 8.2796/8.2797. The first and the second values in the pair refer to the *bid* and the *ask* rates, respectively. The difference between the two values in such a pair is known as the spread. We will use this rate for the following illustrations, and this rate to be provided by service FXDS4.

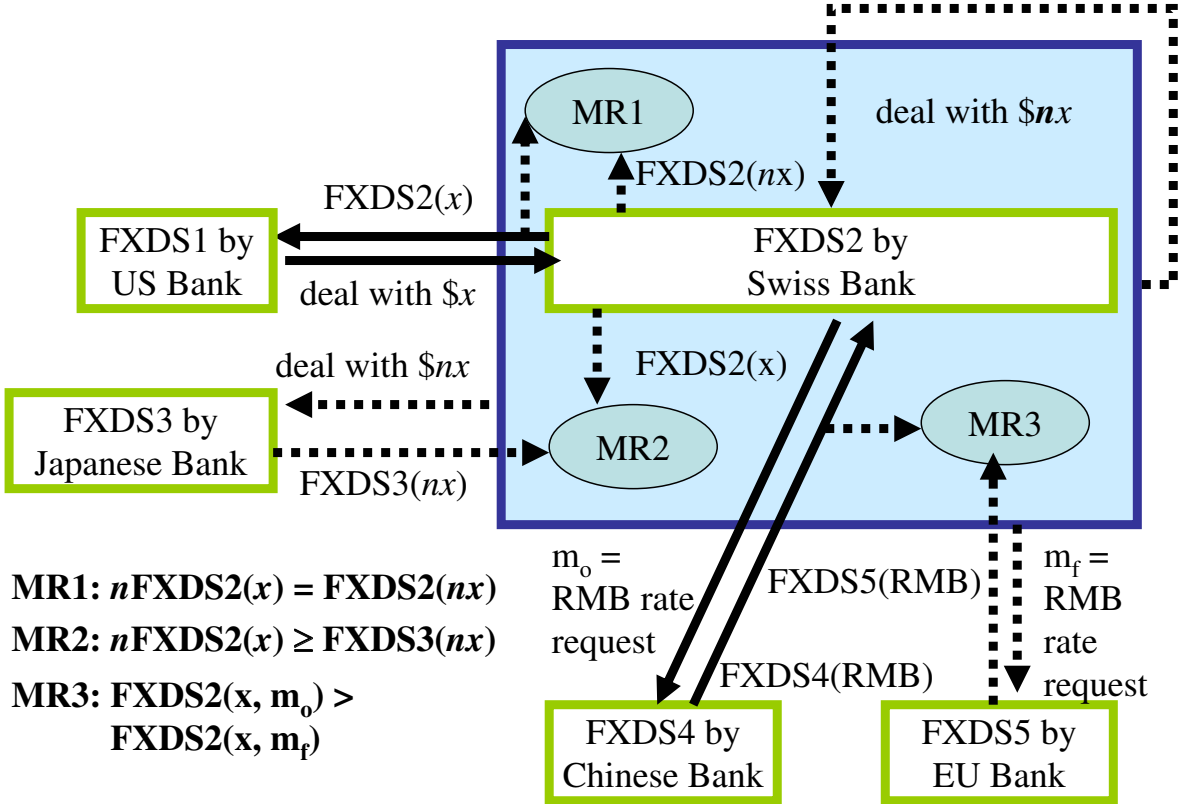

Figure 4: Foreign Exchange Dealing Services

Suppose the expected behaviors of FXDS2 include:

(*i*) It provides a uniform exchange rate for any deal order.

(*ii*) It provides a better, or at least the same, exchange rate to its clients than its rivals (e.g. the service

FXDS3).

(*iii*) It checks the exchange rates from central banks dynamically (e.g. the service FXDS4 or FXDS5).

Further suppose the implementation FXDS2 contains the following two faults:

(*a*) It uses the bid rate or the ask rate to process a deal order nondeterministically.

(*b*) The rate provider logic has faults to cause it to use the minimum (that is, the worst rate) instead of the maximum (that is, the best rate) for its rate calculations.

To test the service FXDS2, testers can apply our testing methodology. The illustrations are in order.

4.2 Unit Testing

Testers first formulate metamorphic relations. For the requirement (*i*), testers can check whether the output amount of service FXSD2 for a deal order is proportional to the deal order size. It forms the first metamorphic relation "MR1: nFXDS2(x) = FXDS2(nx)". Requirements (*ii*) and (*iii*) are related to other services. They are not handled until the integration test phase in this sample scenario.

Consider the (original) test message t_o: a deal order of x (= US$100). FXDS2 correctly uses the above bid rate to output a message FXDS2(US$100) = 827.96 = 8.2796 $\times$ 100. The metamorphic service *MS* constructs a follow-up test case, t_f: a deal order of US$200 = 2 $\times$ x. It then passes this message to FXDS2 to execute. (This is shown as the top-right dotted arrow in Figure 4.) At this time, FXDS2 incorrectly uses the above ask rate and outputs FXDS2(US$200) = 1655.94 = 2$\times$ 827.97. Finally, both messages FXDS2(US$100) and FXDS2(US$200) are checked by *MS* via MR1. We have, 2 $\times$ FXDS2(US$100) = 1655.92 $\neq$ 1655.94 = FXDS2(US$200). It violates MR1. Hence, a failure related to the fault (*a*) is revealed and reported by *MS*.

4.3 Integration Testing

Let us discuss the integration test phase. The support to the integration testing refers to the steps (IT-1) to (IT-7) of the methodology in Section 3.2.

We begin with the formulations of two metamorphic relations in this sample scenario. For the requirement (*ii*), testers may enlarge or shrink a deal order size by a multiplier [1]. Such a follow-up deal order will be forwarded to a rival service (e.g. FXDS3). Next, one can determine whether the output of FXDS2 of an identical order is better than or equal to that from a rival service. This formulates the metamorphic relation "MR2: nFXDS2(x) $\geqslant$ FXDS3(nx)". For the requirement (*iii*), testers may formulate the metamorphic relation "MR3: FXDS2(x,m_o) $>$ FXDS2(x,m_f)", and the like. MR3 means that if the target exchange is between USD and RMB, then the rate, provided by the central bank of China via the rate request m_o, should be strictly better than that due to any other rate request m_f [2].

We then discuss the handling of the follow-up test cases. We will also describe the use of the metamorphic relations to support the integration test phase. We begin with a case that a follow-up test case is generated based on an incoming message of the service S_1. After that, we will describe a case that it is based on an outgoing message of the same service.

The above follow-up test case t_f could be used as an original test case for the integration test phase. According to MR2, a follow-up test case m_f can be formulated: deal order of US\$60 = 0.3 $\times$ 200. Suppose that an implementation of the service FXDS3 is discovered dynamically, and the latter correctly receives m_f and returns a message FXDS3(US\$60) = 60 $\times$ 8.2796 = 496.776 to *MS*. Both messages FXDS2(US\$200) and FXDS3(US\$60) are then passed to the implementation of MR2 for verification. We have, 0.3 $\times$ FXDS2(US\$200) = 496.782 $\geqslant$ 496.776 = FXDS3(US\$60) [3]. It satisfies MR2. Hence, no failure will be reported by *MS* for this particular case.

On the other hand, in the execution of a deal order as the input, FXDS2 needs to communicate with services of central banks to collect relevant exchange rates. Thus, the original test case t_o (or t_f alike) will trigger an outgoing message m_o for such a purpose: a USD/RMB rate request. Suppose that FXDS2 discovers FXDS4 and FXDS5 to provide the quote of exchange rates of RMB. For the

[1] In practice, a deal order size applicable to a global bank may not be applicable to a small foreign exchange shop at a street corner.

[2] We note that we choose to show the internal interactions with other services as parameters in the relation to ease our discussion.

[3] In practice, we need to take rounding into account.

illustration purpose, further suppose that the exchange rate provided by the EU bank via FXDS5 for USD/RMB is 8.2795/8.2798. This spread is wider than that provided by FXDS4, and thus is poorer.

Owing to the fault (*b*), FSDS2 incorrectly selects the later rate to process the deal order. To simplify our subsequent discussion, suppose that the fault (*a*) is corrected. FSDS2 will output a message FXDS2(US$100) = 827.95 = 8.2795 $\times$ 100. *MS* uses the metamorphic relation MR3 to produce the follow-up test case m_f of m_o. Like m_o, this follow-up test case is also a USD/RMB rate request. *MS* then forwards m_f to the service FXDS5 after some service discovery. The rate from FXDS5 is then forwarded to FXDS2. Since there is only one response available, FXDS2 uses it accordingly. So, FXDS2 also outputs 827.95. We have, FXDS2(US$100, m_o) = 827.95 $\not>$ 827.95 = FXDS2(US$100, m_f). It violates MR_3, and *MS* reports a failure due to the fault (*b*).

5 Related Work

Testing research on SOA applications or Web services is relatively new. There are general discussions on Web service testing in [2].

Offutt and Xu [11] propose a set of mutation operators to perturb messages of Web services. They also suggest three rules to produce test cases based on the XML schema of messages. Their initial result on a sample application shows that 78% of seeded faults can be revealed.

Tsai et al. [12] propose an approach to test Web services that each service has multiple implementations with the same intended functionality. They apply test cases to a set of implementations of the same intended functionality progressively. Their test results are ranked by a majority voting strategy to assign a winner as the test oracle. A small set of winning implementations are selected for the integration testing purpose. At an integration testing level, they follow the same approach except using a weighted version of the majority voting strategy instead.

Keckel and Lohmann [8] propose to apply the notion of design by contract to conduct testing for Web services. They suggest defining formal contracts to describe the behavior of functions and interactions of Web services. Based on the contracts, combinatoric testing is suggested to apply to conduct conformance testing against the contracts of intended services.

Our approach uses metamorphic relations to generate follow-up test cases. Those test cases are

not necessary fault-based. This is a major distinction between [11] and ours. Also, a follow-up test case can cause a service to produce an output. The latter can be non-identical to the one due to the original test case. Hence, it can be applied to configurations when multiple implementations of the same functionality are too expensive to be used. Both aspects distinguish ours from [12]. Our approach checks test results amongst themselves; whereas the approach in [8] checks them against some formal contract specifications.

6 Discussion, Conclusion and Future Work

Testing applications with service-oriented architecture needs to deal with a number of issues. They include (*i*) the unknown communication partners until the service discovery, (*ii*) the imprecise black-box information of software components, and (*iii*) the potential existence of non-identical implementations of the same service. In this paper, we treat a service as a software module capable to describe itself, be discover by other services and communicate with other services using well-defined messages and interfaces.

We have introduced a testing methodology to support both unit test and integration test for service-oriented computing. Our work is primarily developed atop metamorphic testing. We formulate the notion of metamorphic services, a kind of access wrapper, a wrapper encapsulates a service under test, implements the metamorphic testing approach. The service approach to conduct integration test alleviates the problem (*i*). It delays the binding of communication partners of the follow-up test cases after service discovery. The metamorphic testing approach alleviates the problems (*ii*) and (*iii*).

The methodology supports testers to apply the test cases for the unit test as the original test case for the integration test. Testers develop metamorphic relations so that the metamorphic services constructs follow-up test cases automatically and test the integration of services autonomously. Hence, services under test can collaboratively delegate the task to conduct integration test to their metamorphic services. Therefore, the methodology has potentials to shift part of the testing efforts from the construction of integration test cases and their expected values to the development of metamorphic relations.

The paper serves as an initial study of the proposed testing methodology. There is quite a number

of follow-up work. First, we have not evaluated our proposal extensively. Experiments are suggested to be conducted. We, in this paper, take it for granted to assume that the computation of the follow-up test cases can be implemented in metamorphic services. Obviously, the real-life practice is more complicated. More detail examinations are needed to realize the testing methodology in the workplaces. The way to control the chain reaction of the follow-up test cases generations due to interferences of multiple metamorphic services will need to be addressed. Last, the degree of coverage of integration test, in the sense of testing criteria, for our novel and dynamic integration testing approach warrant further investigations. There are other questions: How to find out suitable metamorphic relations for a service? Could any metamorphic relation be implemented easily using the metamorphic service approach? When a service under test and a metamorphic relation is given, how to derive a metamorphic service automatically? Is this approach attractive in terms of set up cost when compared to other integration testing approaches? We will report our findings soon.

Acknowledgments

We would like to thank anonymous reviewers, Dr. T. H. Tse and Prof. T. Y. Chen for their useful discussions and comments to improve the paper.

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[2] J. Bloomberg. Testing Web services today and tomorrow.
Available at: http://www-106.ibm.com/developerworks/
rational/library/content/rationaledge/oct02/
webtesting_therationaledge_oct02.pdf, 2002.

[3] W. K. Chan, T. Y. Chen, Heng Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. To appear in *Proceedings*

of the 5th Annual International Conference on Quality Software (QSIC 2005), IEEE Computer Society, Los Alamitos, California, 2005.

[4] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, Polytechnic University of Madrid, Madrid, Spain, 2004.

[5] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45 (1):1–9, 2003.

[6] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 191–195, ACM Press, New York, 2002.

[7] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6 (2):86–93, 2002.

[8] R. Keckel and M. Lohmann. Towards contract-based testing of Web services. *Electronic Notes in Theoretical Computer Science*, 116:145–156, 2005.

[9] M. Mecella and B. Pernici. Designing wrapper components for e-services in integrating heterogeneous systems. *The VLDB Journal*, 10 (1):2–15.

[10] N. K. Mukhi, R. Konuru, and F. Curbera. Cooperative middleware specialization for service oriented architectures. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW 2004)*, pages 206–215, ACM Press, New York, 2004.

[11] J. Offutt and W. Xu. Generating test cases for Web services using data perturbation. *SIGSOFT Software Engineering Notes*, 29 (5):1–10, 2004.

[12] W. T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Hung, and R. Paul. Testing Web services using progressive group testing. In *Proceedings of Advanced Workshop on Content Computing (AWCC 2004), LNCS 3309*, pages 314–322, Springer-Verlag, Berlin, Heideberg, 2004.

[13] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate Web services testing. In *Proceedings of The 7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002)*, pages 171–172, IEEE Computer Society, Los Alamitos, California, 2002.

[14] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 458–466, IEEE Computer Society, Los Alamitos, California, 2004.

[15] W3C. Web Services Activity.
Available at: http://www.w3.org/2002/ws, 2002.

[16] A. Umar. Application Reengineering. Building Web-Based Applications and Dealing With Legacy. Prentice-Hall, Englewood Cliffs, N.J., USA, 1997