Chapter 7.17
A Metamorphic Testing Approach for Online Testing of Service-Oriented Software Applications

W. K. Chan
Hong Kong University of Science and Technology, Hong Kong

S. C. Cheung
Hong Kong University of Science and Technology, Hong Kong

Karl R. P. H. Leung
Hong Kong Institute of Vocational Education, Hong Kong

ABSTRACT

Testing the correctness of services assures the functional quality of service-oriented applications. A service-oriented application may bind dynamically to its supportive services. For the same service interface, the supportive services may behave differently. A service may also need to realize a business strategy, like best pricing, relative to the behavior of its counterparts and the dynamic market situations. Many existing works ignore these issues to address the problem of identifying failures from test results. This article proposes a metamorphic approach for online services testing. The off-line testing determines a set of successful test cases to construct their corresponding follow-up test cases for the online testing. These test cases will be executed by metamorphic services that encapsulate the services under test as well as the implementations of metamorphic relations. Thus, any failure revealed by the metamorphic testing approach

will be due to the failures in the online testing mode. An experiment is included.

INTRODUCTION

The service-oriented architecture (SOA) is an architectural reference model (Bass et al., 2003) for a kind of distributed computing such as the Web services (W3C, 2002). It promises to alleviate the problems related to the integration of heterogeneous applications (Kreger et al., 2003; Mukhi et al., 2004). In this reference model, a SOA application is denoted by a collection of self-contained communicating components, known as *services*. The model also emphasizes that each service should make little or no assumption about its collaborating services. This setting advocates the dynamic composition of a service by using different configurations of supportive services, creating behavioral differences amongst different invocations of a service. Typical end-users of a SOA application, such as bank customers using an online foreign exchange trading service, may expect consistent outcomes each time they use the service. The customers may further compare the online foreign exchange service of a bank to similar services of other banks to judge whether the service is of good quality. If a B2B service provider is driven by a predefined business strategy to, for example, maintain its market share, then the criteria to define functional correctness of the service may vary according to its environment. In other words, testers need to integrate the environment of a service to check test results.

Services may be subject to both the off-line testing and the online testing. Unlike the testing of conventional programs, services bind dynamically to other peer services when it is tested online. While the off-line testing of services is analogous to the testing of conventional programs, the online testing of services needs to address new issues and difficulties.

Testers may generally apply certain static analyses or testing techniques to assure the correctness of a software application. The evaluation criteria of the functional correctness, on the other hand, must be predefined. For a unit testing, testers also want to address the test case selection problem and the test oracle problem (Beizer, 1990). We restrict our attention to the latter problem in this article.

A test oracle is a mechanism that reliably decides whether a test succeeds. For services, as we will discuss, formal test oracle may be unavailable. The expected behavior of a service that represents business goods and services changes according to the environment. Such an expected behavior is relative to the behaviors of competing services or other services. Intuitively, it is hard to define the expected behavior explicitly in the first place. Tsai et al., (2004) for example, suggest using a progressive ranking of similar implementations of a service description to alleviate the test oracle problem. The behaviors of different implementations of the same service vary in general. Test results of a particular group of implementations cannot reliably be served as the expected behavior of a particular implementation of the same service on the same test case. Also, a typical SOA application may comprise collaborative services of multiple organizations, knowing all the implementations are impractical (Ye et al., 2006). For example, a travel package consultant may wrap up services of various hotels, airlines, and entertainment centers to personalize tour packages for a particular client. Without the implementation details, static analysis appears infeasible to assure the implementation quality. The black box approach for test results checking remains viable and popular to assure the correctness of a software application.

Metamorphic testing is a promising approach to the test oracle problem in the testing of conventional scientific programs (Chan et al., 1998; Chen et al., 1998, 2002). Instead of relating an

output to its input, metamorphic testing relates multiple input-output pairs via well-defined relations, called metamorphic relations.

This article extends the preliminary version (Chan et al., 2005a) to propose an online testing approach for testing services. The main contributions of the preliminary version (Chan et al., 2005a) include:

1. It proposes to apply the notion of metamorphic testing to services computing to alleviate the test oracle problem. It constructs more test cases to reveal faults than those ordinarily required when test oracles are known.
2. It proposes to realize the metamorphic testing mechanism as a metamorphic service in services computing that encapsulates a service under test, executes test cases and cross-validates their test results. Such realization seamlessly integrates the existing SOA framework. It automates the construction of follow-up test cases and their test results checking.

The main extended contributions of this article include:

3. It devises the use of the successful test cases for off-line testing as the original test cases for online testing. Since, in an off-line testing, the environment of services can be controlled by testers, test oracle could be defined. Thus, any failure revealed by our metamorphic testing approach will be due to the failures in the online testing mode.
4. It refines the testing approach presented in the preliminary version of this article to take the successful test cases from the off-line testing phase into account.
5. It evaluates the revised proposal against a control experiment having no prior confirmation of test results through an experiment. The experimental result indicates that our proposal is superior to the control experiment. The control approach suffers from an extra 16% effort to check test results and a 13% reduction of failure detection.

The rest of the article is organized as follows. The next section introduces the concepts of services, metamorphic testing and other preliminaries. After that, we describe our testing proposal. Our approach will be illustrated by a sample scenario, followed by an experiment on the feasibility of applying the proposal in a SOA application. Based on the experimental results, we compare our approach to related work, and discuss our experience. Finally, we conclude the article and outline the future work.

PRELIMINARIES

In this section, we briefly introduce the notion of service, metamorphic relation, and metamorphic testing. We also discuss how a metamorphic relation can be aligned with the notion of service and discuss our assumptions. Finally, we clarify our terminologies on online testing and off-line testing.

Service

A *service* in SOA is a self-contained application function, packaged as a reusable component, for the use in a business process (Colan, 2004). It can describe itself so that other services may discover and locate the service. Services also use simple and self-contained messages with well-defined interfaces for communications. Those interfaces are neutral to different hardware or software platforms to support the execution of a service. Meta-models, such as XML Schema (W3C, 2001), are often used to govern the syntactic validity of the messages.

For example, the Web services (Curbera et al., 2002; W3C, 2002) use the *Web service defini-*

tion language or WSDL (W3C, 2005) to let each service of an application define the syntactic and interoperability requirements. It also uses the *universal description, discovery and integration* or UDDI (OASIS, 2005) to offer applications a unified and systematic way to locate services from common registries. The *simple object access protocol* or SOAP (W3C, 2003) is then used to support XML[2]-based messaging between peer services.

In general, there are two types of services, namely *stateless* and *stateful*. They are analogues to conventional programs and object-oriented programs, respectively. Both types are popular in these days.[3] Intuitively, in terms of modeling, one may use a stateless service with self-addressing stateful messages to simulate a stateful service. We thus restrict ourselves to consider stateless services in this article.

Metamorphic Relation (MR)

As a service is a self-contained and function-oriented component, some functional properties of the service should be identifiable; otherwise, it would be difficult for collaborating services to identify the usefulness of the service. One way to display the functional properties of a service is to show the connection of multiple invocations of the service.

A metamorphic relation, or MR, is an existing or expected relation over a set of distinct inputs and their corresponding outputs for multiple executions of the target function (Chen et al., 1998, 2002, 2003). For example, consider a sorting program *Sort*( ) that sorts the input list of integers into a list of integers in ascending order. The expected result of $Sort(\langle 1,4,3\rangle)$ is $\langle 1,3,4\rangle$, which is same as that of $Sort(\langle 1,3,4\rangle)$. Generalizing the scenario would lead to the formulation of the relation: $Sort(X) = Sort(Sort(Y))$ *if* $X = Y$.

Readers can easily observe that in the above example, the expected output of the input $\langle 1,4,3\rangle$ is only induced implicitly. This relieves testers to predetermine the expected behavior of a test case in an absolute term. It is particularly important when a tester is conducting the online testing of a service, for the reason that the concrete expected behavior of a service may depend on other peer services, which may not be under the control of the software developers or testers of the service under test.

A metamorphic relation can be formally described as follows:

> Suppose that f is an implemented function under test. Given a relation r over n distinct inputs $x_1, x_2, ..., x_n$, 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:

$$MR_f = \{(x_1, x_2, ..., x_n, f(x_1), f(x_2), \ldots, f(x_n)) \mid r(x_1, x_2, ..., x_n) \Rightarrow r_f(x_1, x_2, ..., x_n, f(x_1), f(x_2), \ldots, f(x_n))\}$$

We observe that, when a service is seen to be a function, and all input and output messages are self-contained, the notion of metamorphic relation readily applies to services. We will give an example after the introduction of metamorphic testing in the following section.

Metamorphic Testing (MT)

Metamorphic testing or MT (Chen et al., 1998, 2002, 2004) is a program testing technique that employs the mathematical relations, namely metamorphic relations, to conduct testing. It has been applied to numerical applications (Chan et al., 1998) and context-aware applications (Chan et al., 2005c).

It uses *successful test cases* to alleviate the test oracle problem when a test oracle is unavailable or expensive. 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, these test cases will be termed as *successful*. A set of successful test cases is called a successful test set.

Testers may apply the metamorphic testing approach to continue to verify whenever some necessary property of the target function f is satisfied by the implementation P. The metamorphic testing approach constructs the *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 cast an example in the services computing setting. Suppose that S is an expected USD−HKD exchange service, accepting deal orders in USD and returning deal orders in HKD; x and y are two deal orders; $g(\ )$ is a function that accepts a deal order and returns its deal order amount. Further suppose that the following metamorphic relation sample is given, meaning that doubling the deal order in USD, doubling the resultant amount in HKD:

$MR_a(x, y, S(x), S(y))$:
$2g(S(y)) = g(S(x))$ if $g(x) = 2g(y)$

Consider a successful test case x_1 = "a deal order of US$20,000." The metamorphic testing approach constructs automatically another test case 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. As x_1 is successful in the first place, the identified failure should be due to the test case y_1.

In the MT terminology, x_1 is termed as the *original test case*, and y_1 is termed as the *follow-up test case*. By checking the results amongst multiple input-output pairs, metamorphic testing bypasses the need to define the expected result explicitly to alleviate the test oracle problem. We refer readers to (Chan et al., 2005c; Chen et al, 2002, 2003, 2004; Tse et al., 2004) for more details about metamorphic testing.

In this article, we assume that MRs are provided by testers. Verification is generally undecidable, thus, we further assume that the provided MRs describe some necessary conditions of the expected behavior of the program under test. In practice, we recommend testers to define their MRs in a style so that they are directly coded in certain executable specification languages such as Prolog. As such, the implementation of MRs is a non-issue when testers use our methodology. In the next section, we further describe a few major assumptions of our proposal.

Assumptions and Terminologies

In this section, we list out a few major assumptions to establish our model for the online testing of services. These assumptions facilitate us to propose an infrastructure, namely metamorphic service, to be discussed in the metamorphic service section. We also clarify our terminologies on off-line testing and online testing.

We make the following assumptions: A service could be wrapped up by another (wrapper) service. It is based on the understanding that services are loosely coupled amongst themselves and they recognize one another through the common service registries. We assume that in the common service registries, the entries of the former service are replaced by those of the latter service for the testing purpose. This allows the latter service to query and catch messages for the former one. In our model, we use a message as a *test case* or *test result* of a service. This kind of treatment is also adapted by other researchers (Tsai et al., 2004; Offutt & Xu, 2004). It enables the wrapper service to construct test cases and evaluate their test results.

Furthermore, we agree with other researchers that a service is self-contained and independent to the contexts of other services. It also outputs results on every input.

We refer the term *online testing* to the testing of a service under the SOA environment, and the term *off-line testing* to the testing of a service without interaction with other services relevant to the service under test. We also term *off-line testing* and *testing in the off-line mode* interchangeably, and *online testing* and *testing in the online mode* interchangeably. Moreover, for the off-line testing, we refer a *stub service* as a service, since it is obvious in the context that the service under test does not interact with any other peer services of the SOA application when it is subject to an off-line testing. In the next section, we will present our testing proposal.

AN ONLINE TESTING APPROACH

In this section, an online testing methodology will be presented. We propose to test a service in two distinct modes, namely off-line mode and online mode. Informally, we propose to use the test oracle available to the off-line testing and make it also available to the online testing via metamorphic relations. In the overview section, we first introduce the two modes of testing. Next, in the metamorphic service section, we propose a core design artifact of our approach, the metamorphic service, which serves as a surrogate of the services under test to relay messages. At the same time, it constructs follow-up test cases and evaluates results accordingly. It then summarizes the methodology in the testing in the online mode section.

Overview

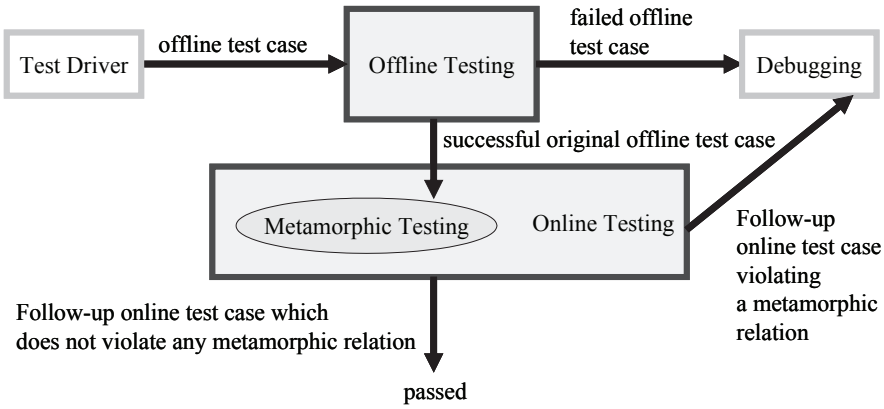
The off-line mode tests a service in the absence of interacting services. We view that it strongly resembles the convention techniques for the unit test of a conventional program. Many test data selection strategies (such as dataflow testing (Beizer, 1990; Zhu et al., 1997)) and test oracle construction for the unit test of conventional programs have been researched for years. As testers could control the off-line environment (such as providing test drivers and stub services, and messages), it is relatively easier to define test sets to test a service and validate the test results in such a mode. Thus, whether a test case reveals a failure or not, for off-line testing, can be determined through the checking of their test results against certain test oracle.

The above observation motivates us to apply the successful test cases of an off-line testing as the original test cases for the online testing mode. In the sequel, we restrict ourselves to discuss the online testing mode. We presume that a set of successful test cases for the off-line testing is available.

For online testing, as these original test cases are determined to be successful in advance, any violation of a corresponding metamorphic relation should be due to the failures of the follow-up test cases. Thus, when their corresponding follow-up test cases are for online testing, our approach pinpoints the online failure-causing inputs automatically even if the test oracle for online testing is not available. Figure 1 depicts the relation between off-line testing and our proposal for online testing.

Interested readers may compare the above strategy and the strategy that an original test case is unknown to be successful. In the latter strategy, when a violation of a metamorphic relation is detected, testers still need to further evaluate whether the violation is due to the original test case (no matter if it is in the off-line or online testing mode) or the follow-up (online) test case. Therefore, this alternative strategy may incur more overheads than our proposal. We will further study this issue in our empirical experiment in the experiment section.

Figure 1. The Integration between off-line and online testing



We will introduce a fundamental building box of our approach in the metamporphic services section, and then elaborate our online testing approach in the rest of this section. To ease our discussion, in the sequel, we use a metamorphic relation with two input-output pairs. The extension of the approach to more than one input-output pair is not hard.

Metamorphic Service (MS)

We formulate a metamorphic service as a service that has the characteristics of being an access wrapper (Mecella & Pernici, 2001; Umar, 1997) and being an agent to conduct metamorphic testing. A metamorphic service imitates all the data and application access paths of the service being encapsulated. It also embodies the program logics, which is the implementation of metamorphic relations, to compute follow-up test cases based on an incoming (or outgoing) message, and evaluates test results according to the implemented metamorphic relations.[4]

Since a metamorphic service is a service, which dynamically discovers a service implementation to receive a follow-up test case. The result of the follow-up test case will be sent to the metamorphic service for the detection of failures. The metamorphic service checks the test results against other test cases by using its metamorphic relations. Any violation of an implemented metamorphic relation reveals a failure.

Testing in the Online Mode

The testing in the online mode validates whether a service can interact correctly with other services. The following shows the self-explanatory methodological steps for this mode.

(ON-1). For a service under test S, collect the set of service descriptions D_S that represents the services interacting with S.
(ON-2). Design a metamorphic relations MR_i applicable to test S in the online mode.
(ON-3). Implement MR_i in the metamorphic service MS of the service S.
(ON-4). Repeat Steps (ON-2) to (ON-3) until no additional metamorphic relation is required for online testing.
(ON-5). For each available successful off-line test case t_o, do
  i. MS uses applicable MR_i to construct the following-up test case t_f of t_o.
  ii. MS invokes S to execute t_f.
  iii. MS obtains the corresponding results t_f

iv. If *MS* detect a failure by using MR_i, then report the failure and go to Step (ON-7).
v. Repeat Steps (On-5-i) to (On-5-iv) until no more applicable MR_i.

(ON-6). Report that no failure is found.
(ON-7). Exit

Step (ON-1) collects the service description that the service under test intends to collaborate. They facilitate testers to define and implement relevant metamorphic relations[5] in Steps (ON-2) and (ON-3) respectively.

If a tester does not identify any further metamorphic relation, Step ON-4 naturally stops, because there is no additional metamorphic relation in the tester's context. On the other hand, if a tester knows a metamorphic relation, but is incompetent to implement such a relation, it is obvious that the tester cannot use such a metamorphic relation according to our methodology; unless the tester seeks helps to implement the metamorphic relation.

Step (ON-5) uses an original test case to construct a follow-up test case. It also invokes the service under test to execute the follow-up test case, and collects the test result of the follow-up test case. Next, it evaluates the results until no more implemented and applicable metamorphic relation is available. When it detects a violation of any metamorphic relation, in Step (On-5-iv), it reports the failure and stops the testing. Otherwise, the process iterates, using another original test case until no more original test case is available. If no failure could be detected by any test case, the process will report such a case in Step (ON-6) and stop at Step (ON-7).

After presenting our approach in this section, the next section will demonstrate a way to use metamorphic testing to reveal failures related to the relative correctness for the testing of a service. We will then present an empirical experiment of our proposal in the experiment section.

AN ILLUSTRATION SCENARIO

In this section, our proposal for online testing will be further illustrated by an example. We first describe an application and its faults. Then, we illustrate how these faults can be revealed.

Figure 2 shows a foreign exchange dealing service application with five services, namely FXDS1 to FXDS5. In particular, FXDS2 is embraced by a metamorphic service. We denote the metamorphic service as *MS*. It has three metamorphic relations, namely MR1, MR2 and MR3. To ease our discussion, we restrict ourselves to discuss the exchange of US dollars to Renminbi.

A bank normally offers cross-currency rates inquiry services. A sample USD−RMB exchange rate is a pair of values such as 8.2796/8.2797. The first value and the second value in the pair refer to the *bid rate* and the *ask rate*, respectively. The difference between the two values in such a pair is known as the spread. We will use this rate for the following illustration, and assume these rates to be provided by the service FXDS4.

Suppose the expected behaviors of FXDS2 include:

1. A uniform exchange rate for any deal order.
2. A better, or at least the same, exchange rate to its clients than its rivals (e.g., the service FXDS3).
3. Checks on the exchange rates from central banks dynamically (e.g., the service FXDS4 for Central Bank of China, or FXDS5 for European Central Bank).

Also suppose that the implementation FXDS2 contains the following two faults (see (Table 2):

a. It uses the bid rate or the ask rate to process a deal order non-deterministically.
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 proposal, which is illustrated as follows.

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).

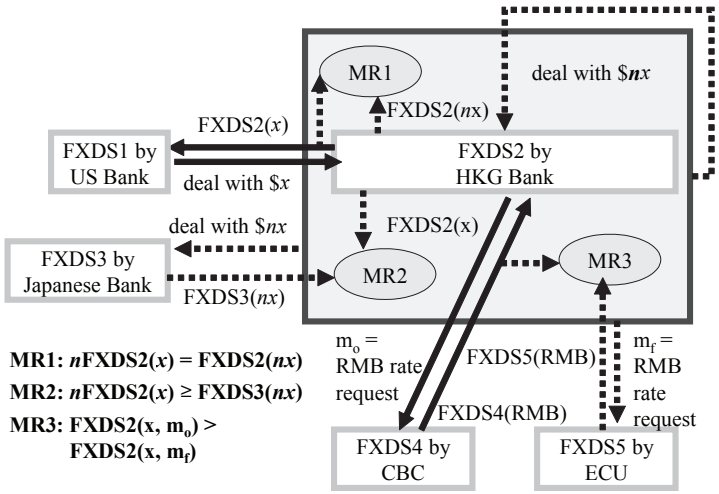
Consider an original test message t_o: a deal order of x (= US$100). FXDS2 correctly uses the above bid rate to output a message FXDS2(US$100) = 827.96 = 8.2796 × 100 in the off-line testing mode by using stubs. The metamorphic service *MS* constructs a follow-up test case t_f: a deal order of US$200 = 2 × x. It then passes this message to FXDS2 to execute in the online testing mode. (This is shown as the top-right dotted arrow in Figure 2).

Suppose, unfortunately, FXDS2 incorrectly uses the above ask rate and outputs FXDS2(US$200) = 1655.94 = 2× 827.97. Since both messages FXDS2(US$100) and FXDS2(US$200) can be checked by *MS* via MR1, we have, 2 × FXDS2(US$100) = 1655.92 ≠ 1655.94 = FXDS2(US$200). It violates the equation MR1. Hence, a failure related to the fault (*a*) is revealed and reported by the metamorphic service.

Readers may be interested to know the results of other combinations of using bid and ask rates for either the original or the follow-up test cases. When the original test case and the follow-up test cases use the bid rate or the ask rate, the equation MR1 will be ineffective to reveal the fault. However, when the original test case incorrectly uses the ask rate and the follow-up test case correctly uses the bid rate, MR1 would reveal the fault.

Let us continue to describe the scenarios to reveal the fault (*b*). We also begin with the formulations of two metamorphic relations. For the requirement (ii), testers may enlarge or shrink a deal order size by a multiplier. (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.) Such a follow-up deal order will be forwarded to a competing service (e.g., FXDS3). Next, one can determine whether the output of

Figure 2. A foreign exchange services system.



FXDS2 of the same order size is better than or equal to that provided by a rival service. This formulates the metamorphic relation:

MR2: nFXDS2(x) ≥ FXDS3(n x).

For the requirement (iii), testers may formulate the metamorphic relation:

MR3: value(FXDS2(x)) > value(FXDS2(y)) if centralbank(FXDS2(x)) = m_o and centralbank(FXDS2(y)) = m_f

Alternatively, in a shorthand notation:

MR3: FXDS2(x, m_o) > FXDS2(x, m_f)

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 from other central banks. We note that we choose to show the outgoing messages that interact with other services as parameters in the metamorphic relation in the shorthand notation to ease our discussion. We also deliberately use a different notation (that is, m_o and m_f, instead of t_o and t_f) for outgoing messages that serve as test cases for the metamorphic testing being illustrated.

According to MR2, a follow-up test case m_f can be formulated: Deal order of US$60 = 0.3 × 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 × 8.2796 = 496.776 to *MS*. Both messages FXDS2(US$200) and FXDS3(US$60) are verified by the metamorphic service via MR2. We have, 0.3 × FXDS2(US$200) = 496.782 > 496.776 = FXDS3(US$60). It satisfies MR2. Hence, no failure will be reported by *MS* for this particular follow-up test case.

On the other hand, for online testing, 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 will trigger an outgoing message m_o, a USD−RMB rate request, for such a purpose. Suppose that FXDS2 discovers both FXDS4 and FXDS5 to provide the quotes of exchange rates for USD−RMB. For the illustration purpose, further suppose that the exchange rate provided by the European Central Bank via FXDS5 for USD−RMB is 8.2795/8.2798. This spread is wider than that provided by FXDS4, and thus is poorer.

MS uses the metamorphic relation MR3 to produce the follow-up test case m_f of m_o. Owing to the faults (*a*) and (*b*), FSDS2 incorrectly selects the ask rate of the service FXDS4 to process the deal order. FSDS2 will output a message FXDS2(US$100) = 827.98 = 8.2798 × 100. We have, in the shorthand notation, FXDS2(US$100, m_o) = 827.96 < 827.98 = FXDS2(US$100, m_f). It violates MR3, and *MS* reports a failure due to the combination of faults (*a*) and (*b*).

We have illustrated the usefulness of using a metamorphic service for the testing of services. In reality, there are domain-specific rounding practices compliant to the international accounting practice. Since rounding rules are a part of the application domain, it is natural for the designed metamorphic relations to take this type of rounding rules into account. This type of domain-specific rounding does not affect the metamorphic approach for services testing. It is worth mentioning that there could be implementation-specific rounding errors also. We refer readers to a previous work on metamorphic testing (Chen et al., 2003) for using a metamorphic approach to identify failures in the presence of the implementation-specific rounding errors. In the next section, we report an experiment that applies our proposal and discuss observations.

EXPERIMENT

In this section, we examine two issues in our proposal through an empirical experiment. We would like to study the testing overhead when some of the original test cases have failed. This is to validate our proposal to use successful test cases applicable to off-line testing as the original test cases for online testing. Our rationale is that if the overhead, on the contrary, is marginal; it is unnecessary to predetermine the successfulness of original test cases. This also helps reduce the testing effort for the online testing of services.

The Subject Program

The subject service implements a service-oriented calculator of arithmetic expressions. It is developed in C++ on Microsoft Visual Studio for .NET 2003 as a Web services. It consists of 16 classes with 2,480 lines of code.

Functionally, the calculator accepts an arithmetic expression consisting of constants of designated data type and arithmetic operators. The set of supported operators are {+,–,×,÷} in which each operator is overloaded to enable every operand belonging to different data types. Five data types of operands are supported: integer, long integer, decimal, floating-point number, and floating-point number with double precision. Each operator is implemented as a supportive service of the subject service.

The subject service parses the inputted arithmetic expressions, locates other services, and composes the results from the results returned by its individual supportive services. In the case where different data types are associated with an operator in a sub-expression, the subject service is responsible to resolve the appropriate data types and to pass the sub-expression to an applicable supportive service. In the design of the application, we choose to allow at most three concurrent instances of the above logic in the subject service. They, together with an expression dispatcher instance, compose our subject service. The expression dispatcher discovers the above supportive services of the subject service, and sends outgoing and receives incoming messages for the latter services. The metamorphic service for the subject service is composed of the dispatcher and a program unit that implements a set of metamorphic relations, which will be explained in the next section.

All the above services are developed by a team of five developers. They have completed certain formal software engineering training. Before doing the experiment, they have gained at least one year of research-oriented and application software development to develop location-based systems, mobile positioning systems and common business applications.

Experimental Setup

In this section, we describe the setup configuration for the testing of the subject service. Specifically, we present the selection of the original test cases, the metamorphic relations, and our experience in finding and implementing the metamorphic relations for the experiment.

As described previously, every instance of services is executed on a designated machine. In total, 14 personal computers are used. All the machines to collect our data are located in the Compuware Software Testing Laboratory of the Hong Kong Institute of Vocational Education (Tsing Yi).

The selection of the set of the original test cases is a black-box combinatorial testing approach. This allows the test cases to serve as the original test cases for both the off-line testing mode and online mode. Consider an arithmetic expression of length 3, which we mean to have an expression having three values and two operators (e.g., 13 + 0.45 – 7). In our experiment, there are five types of data type and four types of operator. In total, there are 2,000 (that is, $5^3 \times 4^2$) combinations, not counting the possible choices of value to initialize

each variable. Since the length of an arithmetic expression could be ranged from zero to a huge figure, we choose to fix the length to three in this experiment. This minimizes the potential problem of using a very long expression to compare with a very short expression in our subsequent analysis. We also initialize test cases that do not cause integer value overflow, and the overflow of other data types alike. We are aware that we have implemented some design decisions in the test case selection process. We minimize these highlighted threats by designing our metamorphic relations *neutral* to these design decisions.

In ideal cases, a user would perceive every calculation as if it is derived analytically using mathematics. However, some expressions would incur rounding errors due to the limitation of programming or system configuration. Different orders of evaluations of an expression could thus produce inconsistent results. A good calculator would provide consistent results to its users. For example, the arithmetic expression "(1 ÷ 3) × 3" would be evaluated as "(1 × 3) ÷ 3" by a good calculator.

We follow the above general guideline to formulate our metamorphic relations. We first describe how we determine and implement metamorphic relations. Next, we will describe the chosen metamorphic relations.

The arithmetic operators for the calculator application domain naturally define associative and commutative properties amongst arithmetic operators. We use these properties as the basis to design our metamorphic relations. This aligns with our objective to design such relations neutral to the selection of the original test cases. Since these properties naturally occur in the domain application, we experience a marginal effort to design associative and commutative relations and convert them into their corresponding formats in the sense of metamorphic relation. Furthermore, since the .NET framework for C++ directly supports the chosen operators, the effort to implement the metamorphic relations in our experiment is also marginal.

The types of metamorphic relation used in the experiment are as follows. Suppose A, B and C are operands of some data type in the set {integer, long integer, floating point number, floating point number with double precision}, and θ_1 and θ_2 are overloaded operators in the set {+, –, ×, ÷}. Consider the expression "(A θ_1 B) θ_2 C." A commutative-property rule to generate the follow-up test cases is: "(A θ_1 B) θ_2 C" $\Rightarrow$ "C θ_2 (A θ_1 B)". It derives the follow-up test case "C θ_2 (A θ_1 B)" based on the original test case "(A θ_1 B) θ_2 C."

Let us denote our subject service by S. The corresponding metamorphic relation of the above commutative rule would be: S("(A θ_1 B) θ_2 C") = S("C θ_2 (A θ_1 B)"). A metamorphic relation derived from an associative rule is S ("(A θ_1 B) θ_2 C") = S("A θ_1 (B θ_2 C)"). We also design variants of the follow-up test case derivation rules to deal with the division operator: "(A × B) ÷ C)" $\Rightarrow$ "(1 ÷ C) × (A × B)," and other similar rules alike. The generation of test cases is implemented as logic in our program.

Let us continue the discussion on the execution of test cases. Even a program is ideal; messages could still be lost or corrupt in a distributed environment. Our subject application does not implement a sophisticated fail-and-retry strategy to handle these anticipated scenarios. To compensate the lacking of this kind of strategy, we choose to abort the execution of a test case when a service returns no output after a timing threshold. After some trials-and-errors in our laboratory environment, we choose the threshold to be 50 seconds per execution of a pair of (original and follow-up) test cases. A typical test case will yield the result with three seconds.

Based on the implementation of the subject application, we create additionally six consecutive faulty versions of the set of supportive services. Each version injects one additional mutation fault to its immediate ancestor. The six consecutive

faults are created in the following order: changing a "+" operator to the operator "–"; changing a "–" operator to the "+" operator'; changing a "×" operator to a "–" operator; swapping the operand of a "÷" operator; changing a "–" operator to the "×" operator; and changing a "×" operator to a "+" operator. These faults support operands of the following data types respectively: floating-point number with double precision, integer, decimal, floating-point number with single precision, long integer, and floating-pointer number with single precision.

The following configurations were deployed for off-line testing and online testing.

- The subject application (the subject service with the original version of the supportive services) simulates an off-line testing environment. The original non-faulty versions of the supportive services serve as stub services for the off-line testing, in the sense of conventional testing. Inexperienced developers implement the subject service and, thus, it naturally contains real faults. Some random test cases do reveal failures from the subject service. We however have reviewed that a set of test cases for the subject service for off-line testing are successful. This set of test cases could be used as the original test cases for online testing according to our approach. We refer this set of test cases to as the set of original test cases in the experiment.
- A faulty application (the subject service with a faulty version of the supportive services) simulates an online testing environment. A faulty version of the supportive services is not identical in behavior to its original counterpart (the test stub used in the off-line testing). The six faulty versions therefore facilitate us to re-use the above set of original test cases yet provide failed test results for some of the elements. This allows us to compare the effect of failed original test cases. In order to avoid biases towards a particular faulty implementation, we put all original test cases and their results of the faulty versions in the same pool for analysis and treat them homogenously.

In total, we executed 22,503 follow-up test cases and also 22,503 original test cases. These two figures are the same because we use metamorphic relations with two input-output pairs in this experiment. For the original version, we execute 3,987 pairs of test cases, and for each faulty version, we execute 3,086 pairs of test cases. To facilitate us to evaluate the effectiveness of test cases, we also mark every test case to be successful or failed, according to the expected result of the arithmetic expression. The result is shown in the next section.

Empirical Results

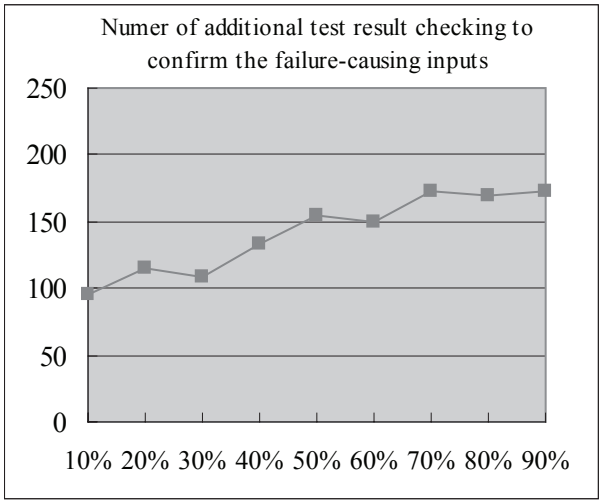
In this section, we will present the empirical results of the experiment and discuss our observations from the results. In summary, our experimental result shows that our approach uses 16% less in terms of effort to detect 13% more in terms of failures, compared to the control experiment having no prior confirmation of test results of original test cases.

We first analyze the overhead, including failed original test cases for online testing. According to the previously mentioned experimental setup, we have collected a pool of 18,516 (= 3,086 × 6) pairs of test cases from the six faulty versions. Some of the pairs contain failed original test cases. We evenly partition the pool into 18 groups so that each group consists of around 1,000 test cases. For the i-th group (for i = 1 to 18), we use database queries to help us randomly draw around 5i percent of its elements having failed original test cases and all elements having successful original test cases to calculate the overhead in terms of test result checking. The total number of elements drawn from each group is shown in Table 1.

Table 1. No. of elements drawn from selected groups

Group	2	4	6	8	10	12	14	16	18	Total
Percentages of failed test cases drawn	10%	20%	30%	40%	50%	60%	70%	80%	90%	-
No. of elements drawn	767	791	809	839	885	904	945	973	997	7,910

Figure 3. The overheads for different percentages of failed original test cases



The calculation of the *overhead* of a test pair is as follows.

- If no violation of the associated metamorphic relation is detected, then the overhead value is zero. It is because the test pair cannot reveal any failure.
- If a violation of the associated metamorphic relation is detected, then the overhead value is equal to the number of actual failure-causing test case(s) in the test pair. In other words, if both test cases are failure-causing inputs, the overhead value will be two; otherwise, it will be one.

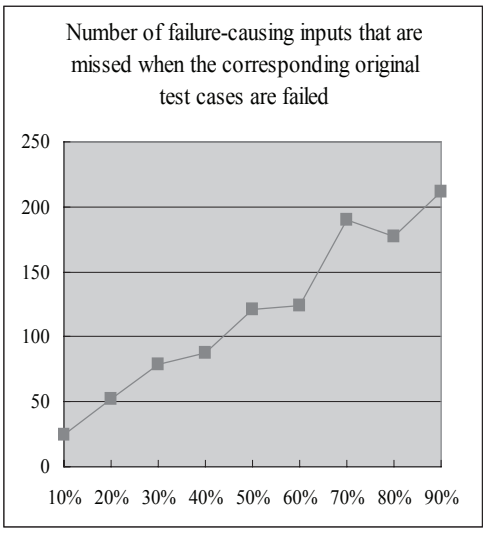
We further define that the number of additional test result checking to confirm the failure-causing inputs for a set of test pairs, denoted by Ω, is the sum of the overhead of every test pair in the set.

Figure 3 shows the value of Ω for 10%, 20% and up to 90% of total number of the failed original test cases of a group for the calculation. The mean value is 141 and the standard derivation is 29.4. (We are aware that there are inadequate numbers of data to calculate a good standard derivation statistically. We show the value just to give readers an impression about the skewness of data set.)

As expected, the value of Ω increases when the percentage increases. When more failed original test cases are added to a set for the calculation of Ω, the chance to detect a failure increases.

However, as we increase the number of test cases from 762 (for group 2) to 997 (for group 18), the change in Ω is moderate and is equal

Figure 4. The number of missed detection of failure-causing inputs



to 77 (= 173 – 96). When around 10% of failed original test cases are included, the value of Ω is 96. This initial value, 96, is even larger than the cumulated change, 77, increasing the percentages from 10% to 90%.

It confirms our assertion that the inclusion of small percentages of failed original test cases for the online testing of a service accounts for a major source of overhead in terms of additional test result checking. This substantiates our recommendation of using successful original test cases only for online testing to alleviate the test oracle problem.

We also observe that the overall percentage is not high. The mean is 15.9%. It suggests that even testers had to include inevitably some failed original test cases; measures should be taken to minimize such an inclusion.

We further examine the chance that we would miss to detect a failure if some failed original test cases are used for online testing. In our experiment, we use simple equations (instead of inequality) as metamorphic relations deliberately. We recall that, in our methodology, an original test case is known to be successful. It follows that any failure detected by the metamorphic testing approach would be due to the failure of the follow-up test case. On the other hand, when no violation of a metamorphic relation (in the form of equation) could be detected, the follow-up test case will be regarded as successful in the experiment.

For a set of test pairs, we define ω as the number of test pairs that each pair consists of one failed original test case and one follow-up test case, and at the same time, the associated metamorphic relation cannot detect the failure. The ω thus measures the missed opportunities to detect failures for online testing.

We also use the data sets reported in Table 1 to calculate ω for each group. The results are shown in Figure 4. The mean value is 118.7, that is, 13% of all failure-causing cases. The corresponding standard derivation is 64.2. The minimum is 25 for Group 2 (that is, the 10% group) and the maximum is 212 for Group 18 (that is, the 90% group).

The trend of missed detection of failures ω is increasing as the percentage of failed original test cases included in a group increases. Moreover, the number of missed detection appears proportional to the percentages of failed original test cases in the data set. This observation looks interesting and warrants further investigations.

RELATED WORK

Literatures on function testing research for SOA applications or Web services are not plenty. Bloomberg (2002) overviews a few major types of testing activity for testing Web services. We refer interested readers to the work of Bloomberg (Bloomberg, 2002) for the overview. In the rest of the section, we review selected related work on providing tools for testing, exception handling testing, generation of test cases and test result evaluation for service-oriented programs.

The use of the testing tool is an indispensable part to automate testing activities. Dustar and Haslinger (2004) propose to develop an experimental testing tool prototype to conduct batch mode and online mode testing. Their tool is reported to expect to handle several types of testing such as functional test and reliability test. Deng et al., (2004) extend AGENDA, a database application testing tool that populates database instances as a part of a test case, to test Web-based database applications. Rather than populating test database instances, SOATest (Parasoft, 2005) also reads data from different data sources to conduct testing. On the Internet, there are quite a few tools to conduct load and stress tests for Web services. Their discussions are not within the scope of this article.

To generate test cases, Offutt and Xu (2004) propose a set of mutation operators to perturb messages of Web services. They also suggest three types of rules to develop 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. These faults include communication faults in SOAP, faults in statements for database query languages and conventional faults. The mutated messages could be considered as a method to construct follow-up test cases in the sense of metamorphic testing. Their empirical study of their approach is valuable. Rather than injecting faults to messages, Looker and Xu (2003) inject faults in SOAP-based programs to test the programs for robustness.

Chen et al., (2004a) also aim at testing for robustness. They focus on testing the exception handling of Java Web services through a data flow coverage approach. They propose to inject faults to system calls to trigger exception throws and require test cases to cover the exception-oriented def-use relations. Through their subject programs are Java-based Web service program; their techniques are not specialized to services testing.

Tsai et al., (2004, 2005) propose an approach to testing 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 is 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. To deal with the test case selection problem, their research group (Tsai et al., 2002a) proposes to generate test cases based on WSDL. Although WSDL-based testing has been proposed for a few years and implemented in testing tools such as SOATest (Parasoft, 2005), their version of WSDL (Tsai et al., 2002b) extends the standardized WSDL (W3C, 2001) to include features to express semantics. It is not difficult to use other voting strategies instead of the majority voting strategy in their approach.

The advantages of semantics checking are also observed by other researchers. An immediate example is our adaptation of metamorphic relation in this article. Keckel and Lohmann (2005), on the other hand, 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, combinatorial testing is suggested to apply to conduct conformance testing against the contracts of intended services.

Our approach uses metamorphic relations to construct follow-up test cases. It is not a fault-based approach; whereas Looker and Xu (2003) suggest a fault-based approach to testing for program robustness. In addition, unlike the work of Offutt and Xu (2004), these follow-up test cases in our approach are intentional to allow automated test result evaluations. A follow-up test case triggers a service to produce an output. The output of a follow-up test case is allowed to be different from the output of the original test case. Hence, it can be applied to configurations when multiple implementations of the same functionality are too expensive to be used. This distinguishes our work from Tsai et al., (2005). Our approach checks test results amongst themselves; whereas the work of Keckel and Lohmann (2005) checks the test results against some formal contract specifications. Chen et al., (2004a) is a white-box-centric approach; whereas ours is a black-box-centric approach. The tools (Dustar & Haslinger, 2004; Parasoft, 2005) appear to be developed as an integrated testing tool that includes a number of different testing modules. Their architectures are unclear to us at this moment. Our metamorphic service is an access wrapper to facilitate off-line and online testing of a service. We aim at reducing the amount of non-core elements from our metamorphic service. In this way, when exposing the function to other services, we would like to develop the approach further so that it enjoys a good scalability.

DISCUSSION

We have presented an approach to online testing and an experiment on a simple application to study metamorphic testing for services testing in the previous sections. The approach is useful for the online testing of a service if testers have difficulties to obtain the expected results of a test case in a cost-effective manner. The test oracle problem has been identified for many years (see also (Beizer, 1990)). In practice, many testers validate their software without a formal test oracle. Metamorphic testing is an approach towards the problem when a metamorphic relation exists and could be identified. We believe that this trend will continue for services testing.

In the rest of this section, we discuss the threat to validity of the experiment. The evaluation of using the metamorphic testing approach for services testing in the online mode is conducted in a small application. The application domain of the subject application is generic that it uses the basic computing functionality, namely a fundamental set of arithmetic operations and basic (communicative and associative) properties to define metamorphic relations. It is unknown about the impact on the results when certain domain-specific knowledge is taken into account to both define metamorphic relations and selections of test cases. We merely use a small number of faulty versions to conduct the experiment to discuss our findings. The statistics may be different if other faulty versions are used. Our subject program does contain real faults and hence some metamorphic relation instances are violated even if the original version is used. We believe that it is common for a typical testing of a newly-coded software program. However, the assumption is not valid if other rigorous quality assurance techniques such as rigorous regressions, code inspections or formal methods have been applied to a software development project. We have evaluated our approach on the Microsoft .NET 2003 platform only. The present implementation of the subject program is not portable to other platforms. There is a threat to interpret the empirical results in other configurations. We have reviewed the set test cases and the set follow-up test cases, and use a commercial spreadsheet to help us to check whether the target relation is maintained in the reviewed samples.

CONCLUSIONS AND FUTURE WORK

Testing services in a services-computing environment 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; (iii) the potential existence of non-identical implementations of the same service; and (iv) the expected behavior of a service potentially depending on the behavior of competing services. In this article, we treat a service as a reusable software module with a well-defined function. A service can introduce itself so that other services can discover and use the service. Services communicate amongst themselves through well-defined messages and interfaces. A message is an input or an output of a service.

We have presented a testing approach to the online testing support of service-oriented applications. We formulate the notion of metamorphic service; the service that has the characteristics of being an access wrapper, the wrapper encapsulates the access for the service under test and implements the metamorphic testing approach. We also propose to use the successful test case for off-line testing as the original test case for online testing, as test oracle is much more likely to be available for off-line testing. Using our online testing methodology, testers builds a bridge between the test oracle available in the off-line testing mode and the test oracle problem encountered in the online testing mode. The services approach to conducting an online test alleviates the problem (i). It delays the binding of communication partners of the follow-up test cases after service discovery. Our realization of the metamorphic testing approach alleviates the problems (ii), (iii) and (iv).

We have also conducted an experiment to evaluate the feasibility of our proposal. The experimental results encouragingly indicate that, on average, when the set of original test cases are unknown to be successful, an extra 16% effort to check test results and a 13% reduction of failure detection are observed. This supports our proposal that original test cases should be (much) better to be successful, particularly when the checking is (much) less costly when it can be conducted in the off-line testing mode.

There are quite a number of future directions of research. We have not evaluated our proposal extensively. We plan to conduct more experiments. The way to control the chain reaction of the follow-up test cases generations due to interferences of multiple metamorphic services warrants more researches. We also plan to measure the degree of code coverage or fault coverage of our approach.

ACKNOWLEDGMENT

We would like to thank the program co-chairs of The First International Workshop on Services Engineering (SEIW 2005) for inviting us to extend the preliminary version (Chan et al., 2005a) to contribute to the special issue. We would also like to thank anonymous reviewer of the article. Special thanks should be given to Mr. Kwong–Tim Chan, Mr. Hoi-Shun Tam, Mr. Kwun-Ting Lee, Mr. Yuk-Ching Lam and Mr. King–Lun Yiu of the Hong Kong Institute of Vocational Education (Tsang Yi) who do the experiments presented in this article. Part of the research was done when Chan was with The University of Hong Kong. This research is supported by grants of the Research Grants Council of Hong Kong (Project Nos. HKUST 6170/04E and CITYU1195/03E).

REFERENCES

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison Wesley.

Beizer, B. (1990). *Software Testing Techniques*. New York:Van Nostrand Reinhold.

Bloomberg, J. (2002). *Testing web services today and tomorrow*. Retrieved from http://www-106.ibm.com/developerworks/rational/library/content/rationaledge/oct02/webtesting_therationaledge_oct02.pdf .

Chan, F.T., Chen, T.Y., Cheung, S.C., Lau, M.F., & Yiu, S.M. (1998). Application of metamorphic testing in numerical analysis. In *Proceedings of IASTED International Conference on Software Engineering (SE 1998)*,(pp. 191–197). Calgary, Canada: ACTA Press.

Chan, W. K., Cheung, S. C., & Leung, K. R. P. H. (2005a). Towards a metamorphic testing methodology for service-oriented software applications, The First International Workshop on Services Engineering (SEIW 2005). In *Proceedings of the 5th Annual International Conference on Quality Software (QSIC 2005)*. Los Alamitos, CA:IEEE Computer Society.

Chan, W. K., Cheung, S. C., & Tse, T. H. (2005b). Fault-based testing of database application programs with conceptual data model. In *Proceedings of the 5th Annual International Conference on Quality Software (QSIC 2005)*. Los Alamitos, CA:IEEE Computer Society.

Chan, W. K., Chen, T. Y., Lu, Heng, Tse, T. H., & Yau, S. S. (2005c). A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the 5th Annual International Conference on Quality Software (QSIC 2005)*. Los Alamitos, CA: IEEE Computer Society.

Chen, F., Ryder, B., Milanova, A., & Wannacott, D. (2004a). Testing of java Web services for robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, (pp. 23–34).New York: ACM Press.

Chen, T. Y., Huang, D. H., Tse, T. H., & Zhou, Z. Q. (2004b). 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)*, (pp. 569–583). Madrid, Spain: Polytechnic University of Madrid.

Chen, T. Y., Tse, T. H., & Zhou, Z. Q. (2002). 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)*. (pp. 191–195). New York:ACM Press.

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

Chen, T.Y., Cheung, S.C., & Yiu, S.M. (1998). *Metamorphic testing: A new approach for generating next test cases* (Tech. Rep. HKUST-CS98-01). Hong Kong: Hong Kong University of Science and Technology, Department of Computer Science.

Chunyang Ye, S.C. Cheung & W.K. Chan (2006). Publishing and composition of atomicity-equivalent services for B2B collaboration. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. Los Alamitos, CA:IEEE Computer Society.

Colan, M. (2004). *Service-oriented architecture expands the vision of Web Services, part 1: Characteristics of service-oriented architecture*. Retrieved from http://www-128.ibm.com/developerworks/webservices/library/ws-soaintro.html.

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

Deng, Y., Frankl, P., & Wang, J. (2004). Testing web database applications, SECTION: Workshop on testing, analysis and verification of web ser-

vices (TAV-WEB) papers. *SIGSOFT Software Engineering Notes*, *29* (5).

Dustar, S., & Haslinger, S. (2004). Testing of service-oriented architectures: A practical approach. In *Proceedings of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (NODe 2004)*, (pp. 97–112). Berlin, Heideberg: Springer-Verlag.

Frankl, P. G. & Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, *14* (10), 1483–1498.

Kapfhammer, G. M., & Soffa, M. L. (2003). A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 2003)* (pp. 98–107). New York: ACM Press.

Keckel, R. & Lohmann, M. (2005). Towards contract-based testing of Web services. *Electronic Notes in Theoretical Computer Science*, *116*, 145–156.

Kreger, H. (2003). Fulfilling the web services promise. *Communications of the ACM, 46* (6), 29–34.

Looker, N., & Xu, J. (2003, October). Assessing the dependability of SOAP RPC based Web services by fault injection. In *Proceeding of IEEE International Workshop on Object-Oriented, Real-Time and Dependable Systems*, Capri Island.

Mecella, M. & Pernici, B. (2003), Designing wrapper components for e-services in integrating heterogeneous systems. *The VLDB Journal*, *10* (1), 2–15.

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

OASIS (2005). *Universal Description, Discovery and Integration* (UDDI) version 3.0.2, retrieved from http://uddi.org/pubs/uddi_v3.htm.

Offutt, J. & Xu, W. (2004). Generating test cases for web services using data perturbation, SECTION: Workshop on testing, analysis and verification of web services (TAV-WEB) papers, *SIGSOFT Software Engineering Notes*, *29* (5).

Parasoft Corporation (2005). SOATest, Retrieved from http://www.parasoft.com/jsp/products/home.jsp?product=SOAP&itemId=101 .

Rajasekaran, P., Miller, J. A., Verma, K., Sheth, A. P. (2004). Enhancing Web services description and discovery to facilitate composition. In *Proceeding of The First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, LNCS 3387, (pp. 55-68). Berlin, Heideberg: Springer-Verlag.

Tsai, W. T., Chen, Y., Paul, R., Huang, H., Zhou, X., & Wei, X. (2005). Adaptive testing, oracle generation, and test case ranking for Web services. In *Proceedings of the 29th Annual International Computer Software and Applications conference (COMPSAC 2005)*, (pp. 101–106). Los Alamitos, CA :IEEE Computer Society.

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

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

Tsai, W. T., Paul, R., Song, W., & Cao Z. (2002b). Coyote: An XML-based framework for Web services testing. In *Proceedings of The 7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002)*, (pp. 173–176). Los Alamitos, CA: IEEE Computer Society.

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

W3C (2003). *SOAP Version 1.2 Part 1: Messaging Framework*. Retrieved from http://www.w3.org/tr/soap12-part1/.

W3C (2002). Web Services Activity, Retrieved from: http://www.w3.org/2002/ws .

W3C (2005). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Retrieved from http://www.w3.org/tr/wsdl20/ .

W3C (2004). *Extensible Markup Language (XML) 1.0 (Third Edition)*. Retrieved from http://www.w3.org/TR/2004/REC-xml-20040204/ .

W3C (2001). *XML Schema*. Retrieved from http://www.w3.org/xml/schema .

Umar, A. (1997). *Application reengineering. Building web-based applications and dealing with legacy.* Cliffs, NJ: Prentice-Hall, Englewood

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Survey, 29* (4), 366–427.

ENDNOTES

1 All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Tel: (+852) 2358 7016. Fax: (+852) 2358 1477. Email: wkchan@cse.ust.hk .

2 XML stands for Extensible Markup Language (W3C, 2004).

3 IBM also proposes to design stateless Web services when it introduces service-oriented architecture (see http://www-128.ibm.com/developerworks/webservices/library/ws-soaintro.html .)

4 We refer readers to Section 0 for the role of a tester in the design and implementation of metamorphic relations, and refer readers to Section 0 for the illustration to construct test cases in the metamorphic approach.

5 Although services are highly recommended to publish their functional properties for other services to judge its usefulness in their areas of concerns and to subscribe to use the service, yet the design of metamorphic relations, the implicit form of functional properties, is not within the scope of this paper. We refer interested readers to a case study (Chen et al., 2004) on the selection of effective metamorphic relations for more details.

This work was previously published in International Journal of Web Services Research, Vol. 4, Issue 2, edited by L. Zhang, pp. 61-81, copyright 2007 by IGI Publishing (an imprint of IGI Global).