MT4WS: an automated metamorphic testing system for web services

Chang-ai Sun*, Guan Wang and Qing Wen

School of Computer and Communication Engineering,
University of Science and Technology Beijing,
Beijing 100083, China
Email: casun@ustb.edu.cn
Email: jayjaywg@qq.com
Email: 616773751@qq.com
*Corresponding author

Dave Towey

School of Computer Science,
The University of Nottingham Ningbo China,
Ningbo 315100, China
Email: dave.towey@nottingham.edu.cn

Tsong Yueh Chen

Department of Computer Science and Software Engineering,
Swinburne University of Technology,
Melbourne 3122, Australia
Email: tychen@swin.edu.au

Abstract: The use of web services has been growing significantly, with increasingly large numbers of applications being implemented through the web. A difficulty associated with this development is the quality assurance of these services, specifically the challenges encountered when testing the applications – amongst other things, testers may not have access to the source code, and the correctness of the output may not be easily ascertained (known as the oracle problem). Metamorphic testing (MT) has been introduced as a technique to alleviate the oracle problem. MT makes use of properties of the software under test, known as metamorphic relations, and checks whether or not these relations are violated. Since MT does not require source code to generate the metamorphic relations, it is suitable for testing web services-based applications. We have designed an XML-based language representation to facilitate the formalisation of metamorphic relations, the generation of (follow-up) test cases, and the verification of the test results. Based on this, we have also developed a tool to support the automation of MT for web service applications. This tool has been used in an experiment to test web services, the evaluation of which is reported in this paper.

Keywords: web services; test oracle; metamorphic testing; automatic testing.

Reference to this paper should be made as follows: Sun, C-a., Wang, G., Wen, Q., Towey, D. and Chen, T.Y. (2016) 'MT4WS: an automated metamorphic testing system for web services', *Int. J. High Performance Computing and Networking*, Vol. 9, Nos. 1/2, pp.104–115.

Biographical notes: Chang-ai Sun is a Full Professor in the School of Computer and Communication Engineering, University of Science and Technology Beijing. Before that, he was an Assistant Professor at Beijing Jiaotong University, China, Postdoctoral Fellow at the Swinburne University of Technology, Australia, and Postdoctoral Fellow at the University of Groningen, The Netherlands. He received his Bachelor's in Computer Science from the University of Science and Technology Beijing, China, and PhD in Computer Science from the Beihang University (BUAA), China. His research interests include software testing and service-oriented computing.

Guan Wang is an MSc student at the School of Computer and Communication Engineering, University of Science and Technology Beijing, where he also received his BSc in Computer Science. His current research interests include software testing and service-oriented computing.

Qing Wen is an MSc student at the School of Computer and Communication Engineering, University of Science and Technology Beijing, where he also received his BSc in Computer Science. His current research interests include software testing and service-oriented computing.

Dave Towey is an Assistant Professor in the School of Computer Science, The University Nottingham Ningbo China, prior to which he was with Beijing Normal University – Hong Kong Baptist University: United International College, China. His background includes an education in computer science, linguistics and languages (BA/MA from the University of Dublin, Trinity College), and PhD in Computer Science from The University of Hong Kong. His research interests include software testing, software design, and technology in education. He is a member of both the IEEE and the ACM.

Tsong Yueh Chen is a Chair Professor of Software Engineering at the Department of Computer Science and Software Engineering in Swinburne University of Technology. He received his PhD in Computer Science from the University of Melbourne; MSc, and DIC in Computer Science from Imperial College of Science and Technology; and BSc, and MPhil from The University of Hong Kong. His current research interests include software testing and debugging, software maintenance, and software design.

1 Introduction

Service-oriented architecture (SOA) has been evolving as a mainstream software development paradigm where web services are the basic elements (Papazoglou et al., 2008; Binder et al., 2009). A web service often implements an application or part of an application, and is able to make a set of operations available to its consumers through the web service description language (WSDL) (W3C, 2012a; Zhang and Yang, 2013). A web service has a specification defined in a standard description, while its implementation can be written in any language. This loosely-coupled nature raises some difficulties for the quality assurance of web services-based applications. For instance, inconsistencies may appear between the specification and implementation, especially when the service is subject to frequent updates due to requirements or infrastructural changes.

Software testing is a widely used quality assurance approach (Zlatev and Brandt, 2007), but the unique features of SOA pose new challenges when testing web services (Canfora and Penta, 2008), in particular, the lack of source code and the restricted control of services limit their testability. Furthermore, these applications may face the test oracle problem – situations where it is impossible or impractical to determine the correctness of programme outputs (Chen et al., 1998). Accordingly, testing web services under SOA requires new approaches, and several new techniques have been developed in recent years (Sharma et al., 2012). However, most existing testing techniques assume the existence of a test oracle, and become inapplicable when the test oracle is absent or too difficult or impractical to apply.

Metamorphic testing (MT) (Chen et al., 1998) was introduced to address the oracle problem. Rather than compare the actual and expected output for a single input, MT makes use of properties of the programme under test to compare the outputs of multiple inputs. Recently, MT has been successfully applied to various application domains, however, these studies are based on some prototype or test scripts specific to the programme under test: there is not yet a general test system or tool to support the automation of MT. This lack of a general MT testing system has hindered the wider adoption of MT in practice.

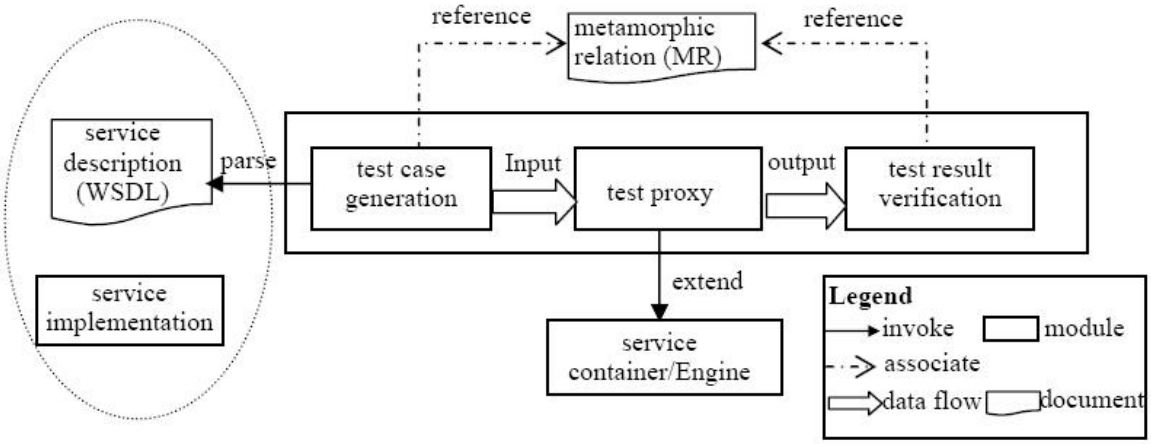
To address the challenges of testing web services and of tooling support for MT, we have developed a MT tool for web services called MT4WS. To our knowledge, MT4WS is the first system supporting automated MT in the area of web services. Furthermore, we have designed a language to facilitate the formalisation of metamorphic relations, the generation of (follow-up) test cases, and the verification of the test results.

The rest of this paper is organised as follows: Section 2 introduces the principles of MT. Section 3 proposes a metamorphic relation description language, MRDL. Section 4 describes the main features, and discusses the architecture and implementation of the MT tool MT4WS. Section 5 describes an empirical study to evaluate the efficiency and effectiveness of MT4WS. Section 6 describes related work, and Section 7 concludes this paper and describes future work.

2 Background

Most testing techniques focus on how to effectively select test cases such that programme faults can be revealed as early as possible, or so that as many faults can be revealed as possible. There is an implicit assumption behind most of these techniques that a systematic mechanism for verifying the test output for any possible programme input exists – i.e., that a test oracle is available and applicable. However, in many practical situations the oracle does not exist, or it is too expensive to use. Such an oracle problem hinders the applicability and effectiveness of many testing techniques.

Figure 1 Framework for MT of web services



MT (Chen et al., 1998) is an innovative approach to alleviating the oracle problem. In MT, users first identify some properties of the programme under test, P, based on which a set of metamorphic relations (MRs) can then be constructed. Traditional testing techniques can be used to generate some test cases, referred to as *source* test cases, which the MRs then convert into *follow-up* test cases. In this study, we only consider those MRs which can be represented as (R, Rf), where R defines the relation between source and follow-up test case, and Rf defines the relation that the corresponding outputs should satisfy. Both source and follow-up test cases are executed, and, instead of using an oracle, the execution results (the outputs) are checked against the MRs – if an MR is violated, a fault is revealed.

As a simple example to illustrate how MT works, consider a programme, P, which finds the shortest path between two nodes, a and b, in an undirected graph, G. For P, we can have an MR that for a graph G, the path from node a to b should be of equal length to the path from b to a. To test P by MT, we generate a source test case (G, a, b), and then construct the follow-up test case (G, b, a). We then execute both (G, a, b) and (G, b, a), and check whether $|P(G, a, b)| = |P(G, b, a)|$, where $|P(G, a, b)|$ denotes the length of the shortest path from node a to node b in G, as calculated by P. If the relation does not hold, then P has a fault.

To address the oracle problem associated with testing web services in the SOA context, an MT framework was proposed as shown in Figure 1 (Sun et al., 2011, 2012). Given a web service, which is composed of a *service description* (in a WSDL file), and a *service implementation* (in any programming language), testers need to identify MRs, based on which they can generate test cases and examine test outputs. MT makes use of two sets of test cases: *source* (generated using any available test case generation techniques) and *follow-up* (derived using the identified MRs and the source test cases). Since web services are often deployed and run in a service container (such as Tomcat) or a service engine, when executing the test cases, a *proxy* is usually required, which can transfer the test cases as input, and return the execution output. When testing web services, all input and output should follow some standards, such as the SOAP protocol (Papazoglou et al., 2008). Finally, testers *verify test results* (i.e., the outputs associated with the source and follow-up test cases) by deciding whether or not the relevant MRs are violated. In summary, the proposed framework combines the basic principles of MT with the unique features of SOA.

Clearly, metamorphic relations play a crucial role in the proposed framework: they are essential in both the generation of follow-up test cases, and the verification of the test results. Development of an automated MT system would depend on a formal representation for MRs, based on which algorithms for test case generation and test result verification could then be designed. In previous studies of MT, testers usually manually described MRs in the code, or embedded them in the test scripts. To our knowledge, there does not yet exist a formal and general description for MRs, the absence of which restricts the automation of MT. We next propose an XML-based formal language to express MRs.

3 Metamorphic relation description language – MRDL

Metamorphic relations are intrinsic properties of a programme, often having the structure "if a source test case and a follow-up test case satisfy …, then their outputs should also satisfy …". We represent such MRs as (R, Rf), where R is the relation between the source and follow-up test cases, and Rf is the relation between the corresponding outputs. To formally express MRs, we have designed a metamorphic relation description language (MRDL) as follows:

---

mrSet ::= (*MR*$^{+}$);

MR ::= (*R*; *Rf*);

R ::= (*relation*; *connectiveRelation**);

Rf ::= (*relation*; *connectiveRelation**);

relation ::= [*NEGOp*] (*expressionFollowUp*;
operator; *expressionSource*);

NEGOp ::= *NOT*;
expressionFollowUp ::= *expDescription*;
expressionSource ::= *expDescription*;
expDescription ::= *var* | *exp*;
var ::= *simpleTypes* | *complexTypes*;
exp ::= *MathematicExp* + *StringExp* | *RelationalExp*
| *ListExp* | *userExp*;
RelationalExp ::= *MathematicExp RelationalOp*
MathematicExp | *StringExp* == *StringExp*;
RelationalOp ::= == | > | < | ≥ | ≤ j | =;
MathematicOp ::= + | – | ∗ | / | %;
StringOp ::= *L* + | *L* – | *REP*;
ListExp ::= *list ListOp list*;
ListOp ::= ⊆;
operator ::= *RelationalOp* | *ListOp* | *userOp*;
connectiveRelation ::= (*logicalConnective*; *relation*);
logicalConnective ::= *AND* | *OR*;

The MRDL is represented in XML because this is what all communications among web services, including both inputs and outputs, are based on. To avoid duplication, both *expressionFollowUp* and *expressionSource* are defined as schema type *expDescription*; and *R* and *Rf* are defined as (*relation*, *connectiveRelation**). From the syntax definitions, we have the following:

- *mrSet* is an n-tuple vector defining a set of MRs for a programme.
- An *MR* defines a metamorphic relation, comprising a relation between the source and follow-up test cases (*R*), and the relation between the corresponding outputs (*Rf*). When *R* and *Rf* are composed of several simple relations, then they should be specified in disjunctive normal form.
- A *relation* is a 3-tuple vector which defines a simple relation consisting of *expressionFollowUp*, *expressionSource* and *operator*.
- Both *expressionFollowUp* and *expressionSource* are expressions composed of variables and operators. Here, variables refer to elements defined in the input or output vectors, and operators refer to symbols that can be used to connect these variables to form a mathematical, relational, or string expression.
- *Operator* defines a relation between the value of *expressionFollowUp* and *expressionSource*. Note that *operator* is the name of a syntactic element in the MRDL, and is different from the operations used to connect variables in a mathematic expression. For example, for numeric variables, addition (+) and multiplication (∗) are operations which can connect variables, while greater than (>) or equal to (=) are *operators* which describe a relation between two numeric values.

Here, *simpleTypes* and *complexTypes* refer to the simple and complex types supported by XML Schema, detailed definitions for which can be found online (W3C, 2012b). *MathematicExp* and *StringExp* refer to mathematic and string expressions, whose syntax is similar to those in the Java language, and in the interests of brevity, will not be discussed further here. *L*+, *L*–, and *REP* refer to adding a character to the start of a string, to the end of a string, and replacing a character in a string. *List* is a complex type supported by XML schema. ⊆ refers to the inclusion of lists, and is defined such that for any L_1 and L_2, if every element of L_1 is also an element of L_2, then $L_1 \subseteq L_2$ is evaluated to be true. *userExp* and *userOp* are reserved expression types and expression operators for future extension. The MRDL definitions above, together with the already defined XML Schema definitions (W3C, 2012b), form a complete syntax for the MRDL.

With the MRDL, it is possible to develop algorithms for automatic test case generation and test result verification, which will be described in the next section.

4 MT4WS design and implementation

We have developed an MT tool for web services named MT4WS. We next give an overview of MT4WS, and then discuss its design and implementation.

4.1 Overview

Although MT is simple to use, and therefore suitable for end users to apply, for the following reasons, an automated system supporting MT is highly desirable:

- Users may need assistance formalising and editing MRs, especially when the MRs are complicated – as was the case with an RMB conversion service (Sun et al., 2012), where their design was time-consuming and error-prone.
- Human input error for MRs may occur, especially when there are multiple MRs. For example, 12 MRs were derived for a seismic service (Sun et al., 2012), which may make it difficult for users to detect inconsistencies as new MRs are identified and entered.
- Follow-up test case generation should be as automated as possible. Follow-up test cases are based on a specific MR, and their manual generation seems both inefficient and potentially difficult, especially when the MRs are complicated.
- Test result verification should be as automated as possible. Similar to follow-up test case generation, test result verification is based on specific MRs, and manually verifying test results can be time-consuming and difficult, especially when the number of test cases is large, or the MRs are complicated.

MT4WS was developed to address these challenges, and to support the automation of MT. Although there is an emphasis on testing web services, it can easily be extended to applications in other areas. MT4WS has the following main features.

- A user-friendly interface enabling interactive and efficient MR input.
- Verification mechanisms to identify contradictions or conflicts among input MRs, which would indicate a faulty or incorrect MR entry.
- Automatic follow-up test case generation and test result verification.
- A highly flexible and configurable control over the testing process.

4.2 Architecture

Figure 2 illustrates the MT4WS tool architecture, comprising four main parts, corresponding to the larger rectangular containers: the *test case generator* is responsible for generating source and follow-up test cases; the *executor* invokes web services with test cases, and intercepts the SOAP responses; the *evaluator* verifies the output of the test cases (both source and follow-up), according to a specific MR, and produces a report after all test case execution has completed; and the *configurator* is used to set options for the testing process, such as which MR to select, and how many test cases to use. Furthermore, the tool stores the MRs and test cases in a database, records all test events in a log file, and produces a test report in a text file or web page.

Although MT4WS was initially designed to support MT of web services, it can easily be extended to applications in other areas. We next examine each component in the architecture individually.

4.2.1 Test case generator

Test case generation is divided into two main parts: generation of source test cases, and generation of follow-up test cases. For source test case generation, firstly, the *WSDL parser* analyses the service description, and determines the appropriate input and output format – the basic input and output vector data types, and, for web services, the composite structure of the messages. We have designed a class to describe the operation format, and we have also reserved an interface to support other applications in the future. Secondly, since source test case generation can be done in several ways [such as random generation, special value generation, iteration generation (Sun et al., 2012), or direct selection from file], an interface (*value generator*) was created to enable selection and implementation of the different methods. Because of its efficiency and lack of bias, random generation was chosen for the current version of MT4WS. Next, the generated values are composed into a SOAP message by the *SOAP composer*, according to the format determined by the *WSDL parser*. Finally, the *MR Verifier* checks the test cases, only selecting those which satisfy the MR's restrictions. An example of such a restriction is shown in Table 1, according to which test cases whose values do not comply with the restriction of $A > 3$ are discarded. The follow-up test cases are constructed when, according to a specific MR, the *transformer* re-assigns one or more variable values in the source test case.

Figure 2 MT4WS architecture

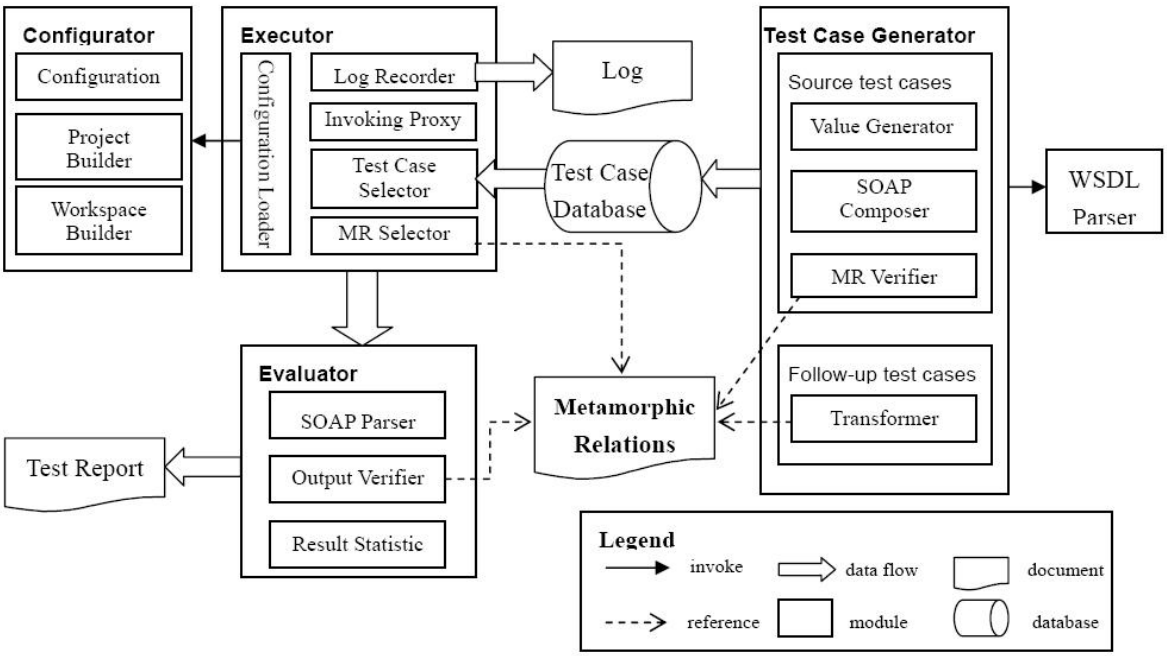


Table 1 MR restriction example

MR	*R*	*Rf*
MR3	A>3 AND B'>B	C'>3C

4.2.2 Executor

The *executor* handles most of the SOA features, and is composed of the following modules:

1 The *invoking proxy* is a reserved interface to call the programme under test, executing source and follow-up test cases. For web services, we use a third party web service API called *soapUI* (Eviware, 2012) as the proxy to invoke web services and intercept their responses. For other types of applications, the proxy should be replaced by the appropriate implementation with consideration of domain features.

2 The *log recorder* is responsible for recording all events during the testing process, and producing a log file.

3 The *configuration loader* loads the customised configuration specified by the *configurator*, and controls the entire test process with the *test case selector* and *MR selector*. Each MR is associated with its own test suite – some MRs may limit source test cases to a special value or a specific range. Only valid test cases can be used as source test cases for each MR.

4 The *MR selector* first selects an MR for testing, and the *test case selector* then selects a test case associated with this MR to apply to the software under test.

4.2.3 Evaluator

The *executor* comprises three modules: *SOAP parser*, *output verifier*, and *result statistics*. Before test result verification, the *SOAP parser* parses the SOAP response from the *executor*, extracting each element's value from the output vector. Based on a specific MR, the *output verifier* verifies source and follow-up test case output, with a test only passing if there is no violation of any relation defined in *Rf*.

4.2.4 Configurator

The *configurator* consists of three modules, and is used to create a testing configuration before tests start. The *configuration* module wraps the user's customised test parameters, including which MRs and test cases are to be used; only the selected MRs are used and appear in the test report. Once testing starts, the *project builder* builds the project, which holds basic information for the web service under test, such as the WSDL and the uniform resource identifier (URI). The *workspace builder* creates folders for the project to save necessary data, such as the MRDL and test report files.

4.3 Implementation

The tool was implemented using Java. In this section, we discuss the key implementation issues.

4.3.1 MR expression

There are two ways to enter MRs into the MT4WS tool: either by importing them directly from an MRDL file or through a stepwise approach which assists users expressing MRs – the tool helps specify the *relations* among key elements (such as *expressionFollowUp*, *expressionSource* and *operator*), and then automatically assembles the basic elements as MRs in the MRDL, finally storing them in an MRDL file.

The tool provides a verification mechanism to check all user-defined MRs and report contradictions: any contradictions would imply either an incorrect MR, or an MR incorrectly entered. For example, if MR_1 and MR_2 are two MRs, where MR_1 is specified as 'if $A' > A$ then $C' > C$", and MR_2 is specified as 'if $A' > A$ then $C' < C$", then MR_1 and MR_2 contradict each other. MRs are saved in the final MRDL file after passing the consistency check.

Currently, as well as all the simple data types defined in the XML schema (W3C, 2012b), the complex data type *list* is also supported. To support additional data types, including complex, user-defined data types, an enhanced version of the MRDL parser would need to be developed. For *R*, due to parsing limitations, when the operator is *AssignOp*, we restrict the *expressionFollowUp* to be only one element in the input vector: the left side of *expressionFollowUp* contains only one of the variables defined in the input vector. Using these three parts (*expressionFollowUp*, *expressionSource* and *operator*), a relationship between the source and follow-up test cases can be defined. If the relationship cannot be adequately described with a single relation, compound relations can be used. Unlike for *R*, *Rf* has no restriction on the number of output vector variables in *expressionFollowUp*. The two remaining parts in *Rf* (*expressionSource* and *operator*) are defined in the same way as for *R*.

When implemented, partly due to the XML schema treating certain characters as special symbols, some symbols or operators are replaced: for example, the single quote symbol (') is replaced by an underscore (_), and '>' is represented by 'GT' (greater than).

4.3.2 Parsing the MRDL

An MRDL file contains a set of MRs for a specific web service operation, and the WSDL URI operation names are defined as attributes in the XML schema. To generate follow-up test cases and verify test results based on MRs in the MRDL, the XML-based MRDL file must first be parsed, a task performed by the MRDL parser in our tool. According to the MRDL file tree structure, MR-related information is stored in a Java tree container, and all element types in the MRDL are processed by a Java class. In this way, parsing an MRDL file is a process of

transforming its contents into Java objects. Among the nodes in the MRDL file, *expressionSource*, *expressionFollowUp* and *operator* are crucial for the generation of follow-up test cases and for test result verification.

4.3.3 Source test case generation

The tool supports three methods of source test case generation:

1 *Manual*: for a specific MR, users may manually input source test cases in an interactive way. Once a test case is entered, the MT4WS verifies that it is valid for the specific MR, and stores it in the database.

2 *Automatic*: MT4WS is also capable of randomly generating valid source test cases for all MRs, requiring only that users specify how many test cases be generated for a specific MR.

3 *Extraction from file*: source test case data can also be extracted from file, in which case the data must follow the MT4WS-defined format, but could be prepared by the testers themselves, or come from previously executed tests.

Figure 3 Follow-up test case generation algorithm

```
INPUT:
  expSrc is an expression of source test case
  inputSrc is an input vector of source test case
  relationOfInput is a relation between source test case and
    follow-up test case
OUTPUT:
  FollowUpTC is the follow-up test case
PROCEDURE transform
1  Foreach (relation in relationOfInput)
2    varFU ← getExpressionFollowUp(relation);
3    expSrc ← getExpressionSource(relation);
4    operator ← getOperator(relation);
5    If (there is a value range on varFU)
6      varFU.value ← genRestrictFollowUpVarValue();
7    Else
8      expSrc.value ← parseExp(expSrc, inputSrc);
9      varFU.value ← genFUVarValue(operator, expSrc.value);
10     while (!operator.verify(varFU, expSrc))
11       varFU.value ← genFUVarValue(operator, expSrc.value);
12   End If
13   FollowUpTC.add(varFU);
14 End Foreach
15 return FollowUpTC;
END PROCEDURE
```

4.3.4 Follow-up test case generation

Once source test cases are available, follow-up test cases can then be generated automatically. To do this, based on the MR representation formalism, we further propose and implement an algorithm for automatic follow-up test case generation, as shown in Figure 3. The algorithm receives a source test case (including its expression *expSrc* and input vector *inputSrc*) and an MR, and outputs a follow-up test case *FollowUpTC*. The algorithm processes each relation in *R* individually. When processing a relation, the value of a variable (or vector) in the follow-up test case is determined by calculating the value of the *expressionSource*, and adjusting it according to the *operator*. When all relations are processed, then all variables of the follow-up test case are calculated and used to construct a complete follow-up test case, which is different from the source test cases only in those variables (or vectors) in the *expressionFollowUp*. Given a set of source test cases, their corresponding follow-up test cases are generated by repeatedly executing the algorithm.

4.3.5 Test execution

The execution of tests in the tool is highly configurable, with options specified by the *Configurator* in Figure 2. A routine is developed to run the entire testing process, during which the tool first selects an MR for testing, and then executes all test cases associated with that MR. When all MRs have been processed, the tool produces a test report.

4.3.6 Verification of test results

For each *relation*, the *output verifier* calculates the *expressionFollowUp* and *expressionSource* associated with the *operator*, and checks whether they satisfy the relation. Considering $R = A' > A$ and $R_f = C' * D' > C * D \quad OR \quad C' > C \quad AND \quad D' < D$, assume the source input vector (A, B) = (1, 2), and follow-up input vector (A', B') = (5, 6); if their outputs are (C, D) = (2, 3) and (C', D') = (0, 1), then this test fails because although $A' > A$ in R is satisfied, $C' > C$ in *Rf* is violated.

Figure 4 shows an algorithm for test result verification, which receives as input the *Rf* of the MR used, and the outputs of the source (*outputSrc*) and follow-up test cases (*outputFU*). The algorithm verifies the outputs against each relation in *Rf*. If all relations are satisfied, it returns true; otherwise, it returns false.

Figure 4 Test result verification algorithm

```
INPUT:
  relationOfOutput is a relation that outputs of source test case and
      follow-up test case should satisfy
  outputSrc is the output of source test case
  outputFU is the output of follow-up test case
OUTPUT:
  pass: true if outputSrc and outputFU satisfy relationOfOutput
PROCEDURE verifyOutput
1  pass = true;
2  Foreach (relation in relationOfOutput)
3    expFollowUp ← getExpressionFollowUp (relation);
4    expSource ← getExpressionSource (relation);
5    operator ← getOperator(relation);
6    expFollowUp.value ← parseExp(expFollowUp, outputFU);
7    expSource.value ← parseExp(expSource, outputSrc);
8    pass ← pass & operator.verify(expFollowUp, expSource)
9  End Foreach
10 return pass;
END PROCEDURE
```

5 System evaluation

In this section, we report on an empirical study conducted to evaluate the MT4WS tool.

5.1 Participants

This study involved six masters-level software engineering students as testers, all of whom had similar software testing knowledge – they had attended software testing courses, and had been introduced to some fundamental principles of MT. Three testers were grouped to form a 'tool team', with each tester assigned a real-life web service to test using MT4WS. The other three testers formed a 'manual team', and each of these testers was also assigned one of the same three web services to test, but they were required to do so by writing their own test scripts. Because there is not yet any comparable MT system, our study was limited to this comparison of our tool with manual testing.

5.2 Measurements

The study applied three objective measurements: efficiency, usability, and applicability. A questionnaire was also used to elicit the 'tool team' testers' subjective evaluation of the tool.

5.2.1 Efficiency

The efficiency of the tool was examined as a comparison of the total time taken to conduct tests either by using the tool or by manually writing the test scripts. Since each approach involves multiple phases, in addition to the total time taken, the time cost in each phase was also calculated. For MT4WS, the entire test process is divided into seven phases, with a separate user interface (UI) corresponding to each. These seven phases are: WSDL parsing; operation selection; MR definition; MR checking; source test case generation method selection; test case preparation; and test case execution and verification. Because the WSDL parsing and the test case execution and verification phases require only very simple user interaction, their time cost was calculated as the execution time. The time taken to select the source test case generation method was negligible, and was therefore not included. For the other four phases, the time cost was measured as the amount of time the user spent on the relevant UI. Manually applying MT to test a web service involved writing a test script to manage the five phases: source test case generation; follow-up test case generation; test case execution; output verification; and results collection. Since the source test case generation time cost for the 'tool team' was omitted, only the manual time costs for each of the other four phases were calculated. All experimental time costs were measured in seconds.

5.2.2 Usability

Usability was measured in terms of the number of mistakes made by the tester: for the 'manual team', a mistake was counted when a tester found and corrected a problem in the test script; and for the 'tool team', a mistake was counted when the tester did something wrong resulting in a warning from the UI, or a testing failure. In both cases, fewer mistakes corresponded to an assessment of better usability.

5.2.3 Applicability

MT4WS was examined from the perspective of how easily it could be applied to test various web services. In this study, three different types of web services (Sun et al., 2012) were selected as subject programmes: the ATM, Seismic, and RMB conversion services. These services differ in terms of their input/output data types, and the complexity of the metamorphic relations – the Seismic service uses the most complex data types, including vectors and lists, and the RMB conversion service has the most complex MRs.

5.2.4 Questionnaire

In addition to the objective measurements outlined above, testers also provided subjective feedback through a especially designed questionnaire for the 'tool team'. The questionnaire consisted of ten multiple-choice questions and two open-ended questions. The multiple-choice questions addressed testers' overall experience using MT4WS, their experience of each testing phase and UI, and their impression of how easy it was to learn how to use MT4WS. Suggestions for additional functionality, or any other comments, were solicited in the open-ended questions.

5.3 Procedure

The experiments involved the following steps:

1 *Training*: Before the experiments, we organised a 30-minute training session to demonstrate the usage of MT4WS to the 'tool team'.

2 *Testing*: For the testing, the 'manual team' members wrote test scripts to generate source test cases, generate follow-up test cases, execute the test cases, verify the output, and collect the results. They recorded the development time spent in each of these phases, and whether or not a test case for a specific MR could kill a mutant. The 'tool team' members used MT4WS to test the web services, recording the time taken in each phase. Both teams generated the source test cases randomly, and used a test suite of one hundred source and one hundred follow-up test cases for each MR – previous studies involving these three subject programmes suggested that this was a sufficient number to ensure the generation of stable results. In this experiment, the testers were provided with previously identified metamorphic relations (Sun et al., 2012) for each of the services under test.

3 *Data collection*: The experimental data (including time cost, and mistake frequency) was collected either by writing appropriate code, as for the 'manual team', or

directly from the testing report, as was the case for the 'tool team'.

5.4 Results and analysis

Table 2 shows the time cost, in seconds, for the 'manual team' for the three subject programmes. From this table, it can be observed that the total time costs for the three subject programmes vary according to their complexity – with the Seismic service being the most complicated, due to the complexity of its data types, and the larger number of MRs.

Table 3 shows the time cost, in seconds, for the 'tool team' for the three subject programmes. From this table, it can be observed that:

1 the total time costs for the Seismic service are still the greatest, but the differences compared with the other two subject programmes are much less

2 for the same subject programme, MR definition and test case execution and output verification were the most time-consuming phases.

Table 4 summarises the time costs for testing each service, either manually or using the tool (MT4WS), and shows that the time savings for tool use over manual testing vary between about 95% and 98%.

Table 5 summarises the mistake frequency for testing each service, both manually, and when using the tool (MT4WS). It was observed that manual testing may incur more human errors, but that using the tool to describe MRs and test cases, because some verification mechanisms are provided, may result in fewer mistakes, and a reduction in the severity of those mistakes. MT4WS could help reduce mistakes by up to 81.25%.

Figures 5 and 6 summarise the survey results for the 'tool team' participants. Each multiple-choice question provided between two and five options, representing the participants' degree of satisfaction with the tool – the highest satisfaction option was one hundred, and scores were evenly distributed between zero and one hundred according to the number of options (e.g., five options meant scores of 0, 25, 50, 75, and 100). The horizontal axis lists different aspects of the tool being measured, and the vertical axis shows the satisfaction score.

Figure 5 MT4WS questionnaire summary part 1: comprehensive evaluation (see online version for colours)

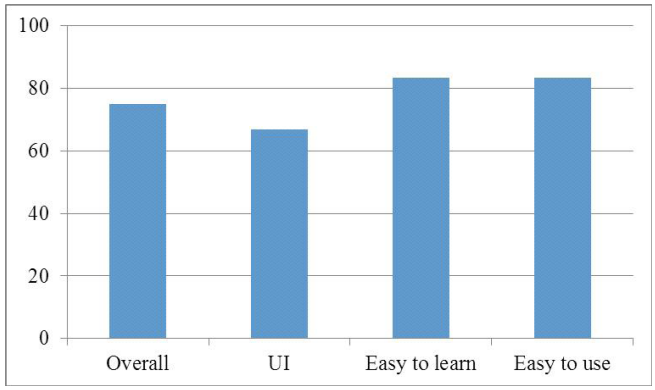


Figure 6 MT4WS questionnaire summary part 2: test process evaluation (see online version for colours)

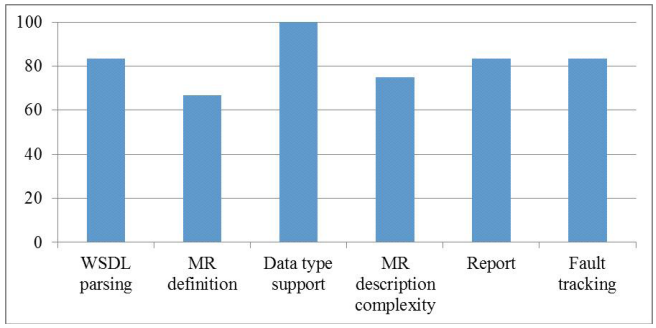


Table 2 Manual time costs for the three subject programmes

	Time cost (s)				
	Follow-up test case generation	*Test scripting and execution*	*Output verification*	*Results collection*	*Total*
ATM	13,572	22,328	8,735	9,286	*53,921*
Seismic	27,300	9,200	32,400	14,400	*83,300*
RMB	12,400	12,300	11,100	4,320	*40,120*

Table 3 Time costs using MT4WS for the three subject programmes

	Time cost (s)							
	Training	*WSDL parsing*	*Operation selection*	*MR definition*	*MR checking*	*Test preparation*	*Test case execution and output verification*	*Total*
ATM	1,800	6	4	36	13	3	157	*2,019*
Seismic	1,800	7	4	78	20	3	141	*2,053*
RMB	1,800	7	2	87	14	3	15	*1,928*

Table 4 A summary of time costs for manual and MT4WS for the three subject programmes

		Total time cost (s)	*Time savings*
ATM	Manual	53,921	96.26%
	MT4WS	2,019	
Seismic	Manual	83,300	97.54%
	MT4WS	2,053	
RMB	Manual	40,120	95.19%
	MT4WS	1,928	

Table 5 Mistake frequency for manual and MT4WS for the three subject programmes

		Frequency	*Improvement*
ATM	Manual	16	81.25%
	MT4WS	3	
Seismic	Manual	17	76.47%
	MT4WS	4	
RMB	Manual	10	80.00%
	MT4WS	2	

The subjective feedback received through the questionnaires reflected a very positive reaction to the tool, and the general opinion that it was of great help in the implementation of efficient and effective MT of web services. Suggestions were also received for the improvement of the UI, in particular for standardising the UI, and making it more user-friendly. It was further suggested that inputting MRs also be made easier.

6 Related work

Web services should be correct and reliable before they are used, and testing is a major activity used to assure their quality. However, the testing of web services is more challenging than that of traditional software due to the unique characteristics of SOA (Canfora and Penta, 2008). In recent years, researchers have proposed various testing techniques to address these challenges (Sharma et al., 2012), and a number of techniques have been proposed for test case generation, including using XML perturbation (Xu et al., 2005), fault injection (Farj et al., 2012), WSDL specifications-based (Bartolini et al., 2009), goal-based (Jokhio, 2009), contract-based (Heckel and Lohmann, 2005), model-driven (Lenz et al., 2007), and ontology-based (Dai et al., 2007). Most testing techniques assume the existence of an oracle, but the testability of web services is often low due to some unique features of the SOA architecture – such as restricted control over the web services and frequent lack of access to their implementation – resulting in the oracle problem.

MT (Chen et al., 1998) has been observed to alleviate the oracle problem (Chen et al., 2011; Liu et al., 2014), and has also detected faults in previously extensively tested, real-life programmes (Xie et al., 2013). A crucial component of MT is the metamorphic relations, since they are used to both generate follow-up test cases, and to verify test results. A key research question for MT is that of how to identify effective MRs, with several researchers proposing guidelines or methods to address this: Asrafi et al. (2011) have observed a correlation between the code coverage achieved by a metamorphic relation and its fault-detection effectiveness. Liu et al. (2012) proposed a simple method for the systematic construction of new metamorphic relations based on already identified ones, observing that the newly constructed relations were very likely to deliver a higher MT cost-effectiveness than the original ones. A recent study (Liu et al., 2014) has also identified the concept of *diversity* as having an important influence on software testing effectiveness – the study determined that a small number of diverse MRs could have a similar fault-detection capability to a test oracle, and could therefore be used as an effective substitute. When selecting MRs to generate test cases, a selection which minimises their similarity would therefore be recommended. These findings are instructive, and can provide very useful guidance when using the MT4WS to test web services.

Another research question for MT relates to the formalisation of MRs, since such a formalism may greatly facilitate the automatic generation of (follow-up) test cases, and the verification of the results. Chan and Cheung (2010) employed MT for online testing of service-oriented software applications, using as source test cases the successful test cases from off-line testing. Castro-Cabrera and Medina-Bulo (2012) proposed using MT to test business process execution language (WS-BPEL) services, deriving MRs from path conditions in the WS-BPEL compositions. Their proposed approach consists of two steps:

1 deriving MRs using machine learning software by Murphy et al. (2008)

2 designing a metamorphic service that wraps the service under test with MRs, as proposed by Chan and Cheung (2010).

In our previous work (Sun et al., 2011, 2012), we proposed an MT framework for web services, and reported on several case studies. Experimental results suggest that the proposed approach not only alleviates the oracle problem, but is also an efficient testing technique in its own right. Unfortunately, there is not yet a general and formal language for description of MRs, which restricts the automation of MT, and hence its efficiency. The MRDL developed in this paper addresses this limitation, and the MT4WS tool has been developed to further improve the automation of MT.

7 Conclusions

MT provides a new approach to alleviate the oracle problem, and has been successfully applied in various application domains. However, most MT case studies are conducted by writing program-specific test scripts. The

absence of a general language for expressing MRs hinders the automation of MT, and consequently there is not yet a general MT support tool. In this paper, we have proposed an XML-based metamorphic relation description language, MRDL, and a set of MT algorithms based on this MRDL. We have developed MT4WS to improve the efficiency of MT. An empirical study was conducted to evaluate the effectiveness and efficiency of MT4WS, and the experimental results show that MT4WS can be used to test web services effectively and efficiently.

In our future work, we would like to further enhance MT4WS by supporting more data and relation types. We would also like to extend the system to support MRs in which follow-up test cases can depend not only on source test cases, but also on their output. Another interesting direction may lie in the incorporation of guidelines to help users to derive metamorphic relations.

Acknowledgements

This research is supported by the National Natural Science Foundation of China (Grant No. 61370061), the Beijing Natural Science Foundation of China (Grant No. 4112037), the Fundamental Research Funds for the Central Universities (Grant No. FRF-SD-12-015A), the Beijing Municipal Training Program for Excellent Talents (Grant No. 2012D009006000002), and an Australian Research Council Grant (ARC DP120104773).

References

Asrafi, M., Liu, H. and Kuo, F-C. (2011) 'On testing effectiveness of metamorphic relations: a case study', in *Proceedings of the 5th International Conference on Secure System Integration and Reliability Improvement (SSIRI 2011)*, pp.147–456.

Bartolini, C., Bertolino, A., Marchetti, E. and Polini, A. (2009) 'WS-TAXI: a WSDL-based testing tool for web services', in *Proceedings of the 2nd International Conference on Software Testing, Verification and Validation (ICST 2009)*, pp.326–335.

Binder, W., Constantinescu, I. and Faltings, B. (2009) 'Service invocation triggers: a lightweight routing infrastructure for decentralised workflow orchestration', *International Journal of High Performance Computing and Networking*, Vol. 6, No. 1, pp.81–90.

Canfora, G. and Penta, M. (2008) 'Service-oriented architectures testing: a survey', *LNCS*, Vol. 5413, pp.78–105, Springer.

Castro-Cabrera, C. and Medina-Bulo, I. (2012) 'Application of metamorphic testing to a case study in web services compositions', in *Proceedings of International Joint Conference on E-Business and telecommunications (ICETE 2011)*, pp.168–181.

Chan, W.K. and Cheung, S.C. (2010) 'A metamorphic testing methodology for online SOA application testing', in Zhang, L.J. (Ed.): *Web Services Research for Emerging Applications: Discoveries and Trends*, Chapter 3, IGI Global, Hershey, PA.

Chen, T.Y., Cheung, S.C. and Yiu, S.M. (1998) *Metamorphic Testing: A New Approach for Generating Next Test Cases*, Technical report, hkust-cs98-01, Hong Kong University of Science and Technology.

Chen, T.Y., Tse, T.H. and Zhou, Z.Q. (2011) 'Semi-proving: an integrated method for program proving, testing, and debugging', *IEEE Transactions on Software Engineering*, Vol. 37, No. 1, pp.109–125.

Dai, G., Bai, X., Wang, Y. and Dai, F. (2007) 'Contract-based testing for web services', in *Proceedings of 31st IEEE Annual International Computers Software and Applications Conference (COMPSAC 2007)*, pp.517–526.

Eviware (2012) *soapUI: a Web Services Testing Tool* [online] http://www.soapui. org (accessed 27 September 2012).

Farj, K., Chen, Y. and Speirs, N.A. (2012) 'A fault injection method for testing dependable web service systems', in *Proceedings of 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2012)*, pp.47–55.

Heckel, R. and Lohmann, M. (2005) 'Towards contract-based testing of web services', in *Proceedings of the 2004 International Workshop on Test and Analysis of Component Based Systems (TA-CoS 2004)*, pp.145–156.

Jokhio, M.S. (2009) 'Goal-based testing of semantic web services', in *Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pp.707–711.

Lenz, C., Chimiak-Opoka, J. and Breu, R. (2007) 'Model driven testing of SOA-based software', in *Proceedings of the Workshop on Software Engineering Methods for Service-Oriented Architecture (SEMSOA 2007)*, pp.99–110.

Liu, H., Kuo, F-C., Towey, D. and Chen, T.Y. (2014) 'How effectively does metamorphic testing alleviate the oracle problem?', *IEEE Transactions on Software Engineering*, Vol. 40, No. 1, pp.4–22.

Liu, H., Liu, X. and Chen, T.Y. (2012) 'A new method for constructing metamorphic relations', in *Proceedings of the 10th International Conference on Quality Software (QSIC 2012)*, pp.59–68.

Murphy, C., Kaiser, G., Hu, L. and Wu, L. (2008) 'Properties of machine learning applications for use in metamorphic testing', in *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE 2008)*, pp.867–872.

Papazoglou, M., Traverso, P., Dustdar, S. and Leymann, F. (2008) 'Service-oriented computing: a research roadmap', *International Journal of Cooperative Information Systems*, Vol. 17, No. 2, pp.223–255.

Sharma, A., Hellmann, T. and Maurer, F. (2012) 'Testing of web services – a systematic mapping', in *Proceedings of 8th IEEE World Congress on Services (SERVICES 2012)*, pp.346–352.

Sun, C.A., Wang, G., Mu, B.H., Liu, H., Wang, Z.S. and Chen, T.Y. (2011) 'Metamorphic testing for web services: framework and a case study', in *Proceedings of the 9th International Conference on Web Services (ICWS 2011)*, pp.283–290.

Sun, C.A., Wang, G., Mu, B.H., Liu, H., Wang, Z.S. and Chen, T.Y. (2012) 'A metamorphic relation-based approach to testing web services without oracles', *International Journal of Web Services Research*, Vol. 9, No. 2, pp.51–73.

W3C (2012a) *Web Services Glossary* [online] http://www.w3.org/TR/ws-gloss/ (accessed 27 September 2012).

W3C (2012b) *XML Schema Part 0: Primer*, 2nd ed. [online] http://www.w3.org/ TR/xmlschema-0/#CreatDt (accessed 27 September 2012).

Xie, X., Wong, E., Chen, T.Y. and Xu, B. (2013) 'Metamorphic slice: an application in spectrum-based fault localization', *Information and Software Technology*, Vol. 55, No. 5, pp.866–879.

Xu, W., Offutt, J. and Luo, J. (2005) 'Testing web services by xml perturbation', in *Proceedings of 16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, pp.257–266.

Zhang, L. and Yang, Y. (2013) 'Dynamic web service selection group decision-making method based on hybrid QoS', *International Journal of High Performance Computing and Networking*, Vol. 7, No. 3, pp.215–226.

Zlatev, Z. and Brandt, J. (2007) 'Testing the accuracy of a data assimilation algorithm', *International Journal of Computational Science and Engineering*, Vol. 3, No. 4, pp.305–313.