Software Qual J (2017) 25:841–869
DOI 10.1007/s11219-016-9337-3

Application of metamorphic testing monitored by test adequacy in a Monte Carlo simulation program

Junhua Ding[1] · **Xin-Hua Hu**[2]

Published online: 14 September 2016

Abstract One of the grand challenges for adequately testing complex software is due to the oracle problem. Metamorphic Testing (MT) is a promising technique to alleviate the oracle problem through using one or multiple Metamorphic Relations (MRs) as test oracles. MT checks the satisfaction of every MR among the outputs of the MR-related tests instead of the correctness of individual test outputs. In practice, it is fairly easy to find MRs for testing any program, but it is very difficult to develop "good" MRs and evaluate their adequacy. A systematic approach for developing MRs and evaluating their adequacy in MT remains to be developed. In this paper, we propose a framework for evaluating MT and iteratively developing adequate MRs monitored by MT adequacy evaluation. The MT adequacy is measured by program coverages, mutation testing, and testing MRs with mutation tests. The MT evaluation results are used for guiding the iterative development of MRs, generating tests, and analyzing test outputs. We explain the framework through a testing example on an image processing program that is used for building the 3-dimensional structure of a biology cell based on its confocal image sections. In order to demonstrate the effectiveness of the proposed framework, we reported a case study of testing a complex scientific program: a Monte Carlo modeling program that simulates photon propagations in turbid tissue phantoms for accurate and efficient generation of reflectance images from biological tissues. The case study has shown the effectiveness of proposed MT framework for testing scientific software in general and the necessity of the MT enhancement in the development of adequate MRs. The case study results can be easily adapted for testing other software.

Keywords Oracle problem · Metamorphic testing · Metamorphic relation · Test coverage criterion · Mutation testing · Monte Carlo method

✉ Junhua Ding
dingj@ecu.edu

[1] Department of Computer Science, East Carolina University, Greenville, NC, USA

[2] Department of Physics, East Carolina University, Greenville, NC, USA

1 Introduction

While adequately testing complex software is challenging, testing "non-testable" software is more difficult due to the oracle problem (Barr et al. 2015; Weyuker 1982). Many scientific software systems belong to the "non-testable" category. In this paper, we will explain a new approach and demonstrate its effectiveness for adequately testing "non-testable" programs with results of study performed on two scientific software systems. Scientific software typically contains a large computational component for supporting scientific investigation and decision making (Kanewala and Bieman 2014). The examples of scientific software include simulation software of nuclear reactions, software for predicting and tracking hurricanes, and software for analyzing medical images. Testing scientific software faces many challenges due to its complexity and the knowledge gap 1: public int getMid between the software developer and its tester (Kanewala and Bieman 2014). The oracle problem is the key to solve the testing of scientific software (Kanewala and Bieman 2014). Many techniques have been developed for alleviating the oracle problem. The techniques can be classified into three categories: the references-based technique that checks a test against a different implementation of the program, reference data sets, experiment results, and any other available references (Sanders and Kelly2008; Weyuker 1982); the analytics-based technique that checks a test against analytics results and special test cases (Mayer 2005; Farrell et al. 2011; Nguyen-Hoan et al. 2010; and Metamorphic Testing (MT) (Chen et al. 1998; Segura et al. 2016) There is no perfect solution to solve the oracle problem, but MT provides a framework that can be easily integrated with any other technique for developing test oracles to adequately test scientific software (Jameel et al. 2015). MT aims at verifying the satisfaction of every metamorphic relationship (MR) among the outputs of the MR-related tests instead of the correctness of each individual output (Chen 1998). If an MR is violated, the software under test (SUT) must have defects (Chen 1998; Murphy 2009). Specifically, MT creates tests according to MRs and then verifies the predictable relations among the outputs of the MR-related tests. Given $f(x)$ as the output of test input x and t as the transform function for an MR, one can create an MR-related metamorphic test input $t(x, f(x))$ by applying function t to source test input x. The transform allows testers to predict the relationship between the outputs of source test input x and its transformed follow-up test input $t(x, f(x))$ according to MR: $(f(x), f(t(x, f(x))))$ (Xie et al. 2011). MT was first proposed by Chen for test generation and alleviating oracle problems (Chen et al. 1998). It has been successfully applied for testing many different applications such as bioinformatics, machine learning, compilers, scientific computing, large-scale database, and online service systems (Chen et al. 2009; Ding et al. 2010; Ding and Zhang 2016; Le et al. 2014; Lindvall et al. 2015; Segura et al. 2016; Xie et al. 2011; Zhou et al. 2015).

Results was collected from applications, and empirical studies of MT show that the effectiveness of MT is highly dependent on the quality of the identified MRs and tests generated from the MRs. The violation of an MR implies defects in the software under test (SUT), but satisfaction of the MR does not guarantee the absence of defects. Program *getMid* shown in Fig. 1 is implemented to find the middle value among three inputs. This program has a permutation property, which means the output is independent of the order of the three inputs. For instance, two inputs $<x1, x2, x3>$ and $<x1, x3, x2>$ should produce the same output. Thus, the permutation property can be used as an MR for testing *getMid*. If a faulty version of *getMid* that misses the code section from line 13 to line 17 was implemented, the defect can be easily revealed by MR *permutation* with metamorphic test ($<x1,$

Fig. 1 Program *getMid*

```
1:  public int getMid
        (int x1,int x2,int x3){
2:     int t = 0;
3:     if ( x1 > x2){
4:       t = x1;
5:       x1 = x2;
6:       x2 = t;
7:     }
8:     if ( x2 > x3){
9:       t = x1;
10:      x2 = x3;
11:      x3 = t;
12:    }
13:    if ( x1 > x2){
14:      t = x1;
15:      x1 = x2;
16:      x2 = t;
17:    }
18:    return x2;
19: }
```

x2, x3>, <*x1, x3, x2*>), since they return two different middle values. Nevertheless, incorrect outputs of a faulty program could accidently satisfy an MR. For example, if line 18 in *getMid* is implemented to return *x1* instead of *x2*, then the defect will not be found by MR *permutation*. It is worth noticing that the defect is not necessarily revealed even all statements or branches of the program are covered by the tests. Although MT can be enhanced with structural testing or other white-box testing techniques, MT evaluation requires additional test adequacy criteria that are specifically designed for the evaluation of adequacy of MRs. The criteria for evaluating the adequacy of MRs and the approach for iteratively creating adequate MRs guiding by test evaluation results are two topics to be addressed in this research.

In this paper, we propose an enhancement to MT with a framework for guiding the iterative development of adequate MRs and tests based on MT test adequacy evaluation results. The test evaluation consists of three tasks: evaluation of program coverage such as statement coverage and condition coverage; mutation testing; and testing MRs with mutation tests. A mutation test set is a group of tests that violate an MR that is under evaluation. The program coverage is evaluated through runtime code instrumentation. Mutation testing (Pezzè et al. 2007) and testing MRs with mutation tests are conducted separately. The test evaluation result is used as a reference to examine the adequacy of tests and MRs, guide iterative test generation and MR development, and analyze defects in the SUT. The program coverage criteria that are evaluated in this research include: *statement coverage*, *definition-use pair coverage*, *function coverage*, *branch coverage*. Statement coverage measures the percentage of statements that are executed by a test suite. Definition-use pair coverage measures the percentage of definition-use pairs that have been exercised by a test suite to all definition-use pairs in the SUT. Function coverage measures the percentage of functions that are executed by a test suite to all functions in the SUT. Branch coverage measures the percentage of branches from all decision points that are executed by a test suite to all branches in the SUT. Mutation testing measures the percentage of mutants that are killed by a test to all mutants that are instrumented in the SUT. Because the correctness of an individual output of a "non-testable" program is unknown, mutation testing suffers more serious problem of the equivalent of mutants in MT. It is

much more difficult to determine whether a mutant is killed by an MR in MT. Therefore, the status of "weakly killed" is defined in this paper. If the outputs of the original SUT and the mutated one of a test are different, the mutant is counted as weakly killed even it was not killed by any MR. Testing MRs with mutation tests means each MR is tested with tests that violate the relation. It is a type of negative testing for evaluating the quality of MRs. For each MR, at least one mutation test set that violates the MR is created and tested. The adequacy of tests means each test adequacy criterion is 100 % covered by the tests, and the adequacy of MRs means the MRs are able to produce adequate tests. Although the subsumed relations exists among statement coverage, function coverage, and branch coverage, the coverage evaluation results in different criteria that can be used for test analysis at different levels. For example, the adequacy of the branch coverage subsumes the adequacy of the function coverage, but the adequacy of the function coverage often offers a quick feedback of the test effectiveness. Several techniques for evaluating the adequacy of program coverage have been reported (Ding et al. 2009); (Zhu et al. 1997); (Zhu and He 2002). In this paper, program coverage is automatically evaluated at runtime, but both mutation testing and testing MRs with mutation tests are evaluated manually. We explain the process and idea of the proposed MT technique monitored by test adequacy evaluation in an example of testing on a medical image processing program, which is a small "non-testable" scientific program for constructing the 3-dimensional (3D) structure of a biology cell based on a stack confocal image sections of the cell (Ding et al. 2010). The effectiveness of the technique is further demonstrated by a case study of testing a large scientific software system: a Monte Carlo simulation program for modeling photon propagation in biological tissue for accurate and efficient generation of reflectance images from turbid tissue phantoms (Chen et al. 2007).

The contributions of this paper are summarized as follows: (1) Developed a framework for the enhancement of MT with demonstration through an example. The framework includes a set of test adequacy criteria and a procedure for evaluating the adequacy of MT and an approach for iteratively developing adequate MRs and tests guided by test adequacy evaluation results. (2) Conducted a case study of the framework on a Monte Carlo simulation program, which is a representative "non-testable" scientific program with common characteristics of scientific software. The framework can be easily extended for testing similar programs. We discovered a defect in the program and found the cause guided by the test adequacy evaluation result. The case study results demonstrated that the MT monitored by test adequacy is an effective approach for testing scientific software. (3) The approach and process for the development of MRs, test adequacy evaluation, and test generation proposed in this paper can be used for testing other complex software systems.

The rest of the paper is organized as follows. In Sect. 2, we present the MT monitored by test adequacy with a running example, which is an image processing program. In Sect. 3, we describe the case study of testing a Monte Carlo simulation program using the proposed MT. The related work is discussed in Sect. 4, and the paper is concluded in Sect. 5.

2 MT monitored by test adequacy

In this section, we explain the procedure of the MT monitored by test adequacy through testing an image processing program for reconstructing the 3D structure of a biology cell based on its confocal image sections. The confocal image sections are taken along z-axis

with a step size of 5 μm between image sections using a confocal microscope. A confocal microscope places a spatial pinhole at the back focal plane of the objective lens to eliminate out-of-focus light so that the depth of field of the image will be extremely short less than 1 μm in order to acquire high resolution and high contrast images, which enable the reconstruction of the 3D structure from the obtained images (Confocal microscope et al. 2016). The cell is stained with two fluorescence dyes so that two cellular organelles of nucleus and mitochondria are shown in different color channels. For example, the nuclear is in red, and cytoplasm is in light green and mitochondria are in bright green as shown in Fig. 2. Each image section is processed with pattern recognition algorithms to segment the stained organelle such as nucleus, cytoplasm, and mitochondria from each other. The 3D structure of a cell is reconstructed based on the processed image sections. The morphology parameters of nucleus, cytoplasm, and mitochondria are calculated based on the reconstructed 3D structure. Figure 2 shows a sample input and output of the program, where the input showing in the left panel consists of several confocal image sections of a cell, the one showing in the right bottom is a segmented image section with red for nucleus, blue for mitochondria, and green for cytoplasm, and a sectional view of a 3D reconstructed structure of a cell is shown at top right. We use the software component for processing mitochondria and building 3D structures of mitochondria in the program as an example here to explain the MT procedure. As shown in Fig. 3, the testing process consists of an iterative sequence of activities including *identification of MRs*, *test generation*, *test executing, and test adequacy evaluation*.

2.1 Identification of MRs

Testing the software component for reconstructing 3D structures of mitochondria in the image processing program is complex because each cell has many mitochondria in different shapes and sizes. It is hard to find a test oracle to check whether a reconstructed 3D structure of a mitochondrion is same as those of a real cell structure. Guided by the prior experiences (Mayer and Guderlei 2006; Murphy et al. 2008; Murphy et al. 2009; Xie et al. 2011) and domain knowledge in biomedical image processing, we first identify an initial set of MRs and then create a set of initial test (or called source tests), which are a set of confocal image sections. Based on the identified MRs and source tests, follow-up tests are created through editing mitochondria in confocal image sections of the source tests. Paired source tests and follow-up tests are used together to test the program through checking the satisfaction of each MR. Six MRs are first identified based on the general guideline for identifying MRs (Mayer and Guderlei 2006; Murphy et al. 2008):

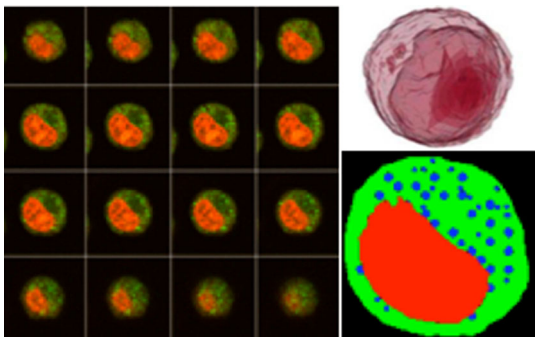

Fig. 2 Left panel is confocal image sections of a cell. The *top right* image is a sectional view of a 3D cell structure, and the *right bottom* one is a processed image section

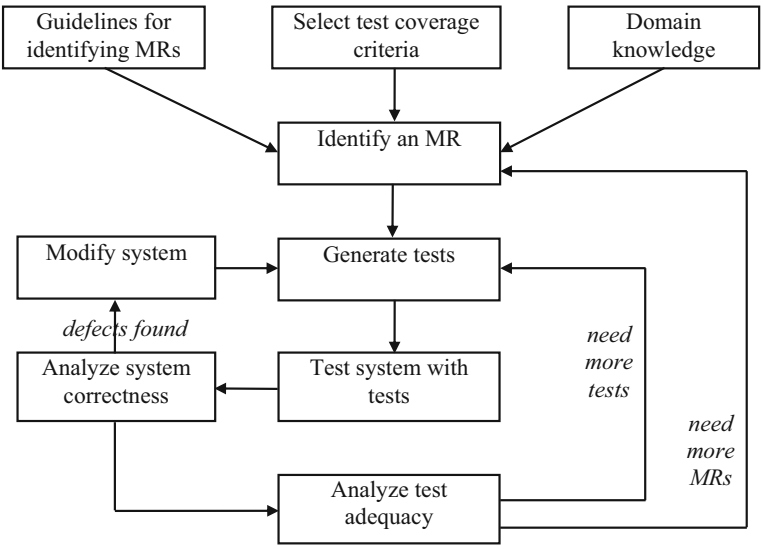

Fig. 3 Process of the metamorphic testing monitored by test adequacy

MR1: Inclusive The program first draws the contour line of each mitochondrial cluster containing one or multiple mitochondria in every confocal image section and then builds a 3D structure through connecting the contour lines of the same mitochondrial cluster in adjacent image sections. If an artificial mitochondrial cluster is added to original confocal image sections, the added one should be recognized and constructed, the total number of mitochondrial cluster should be increased by one, and the calculated volume of the mitochondria in the cell should be increased correspondingly. We first add one artificial mitochondrial cluster to only one image section, and then the same mitochondrial cluster can be added to multiple image sections (i.e., the mitochondrion is sufficiently large that it crosses multiple image sections). Modification of the input images is completed with a MATLAB code, and each modified image is visually inspected to ensure that it does not produce unexpected results.

MR2: Exclusive A mitochondrial cluster is removed from the original confocal image sections. Since the same cluster may appear in multiple adjacent image sections, it has to be completely removed from all of these image sections. Then, the removed cluster should not appear in the processed image sections or in the reconstructed 3D structure output. The total number of the mitochondrial clusters should be decreased by one, and the volume of mitochondria is expected to be decreased.

MR3: Multiplicative The size of a mitochondrial cluster is enlarged with a small percentage from the original confocal image sections. Since the same cluster may appear in multiple image sections, the same percentage of increasing is applied to the cluster at every image section. Then, we calculate whether the size of the mitochondrion is increased in the processed image sections and in the reconstructed 3D structure output. The total number of the mitochondrial clusters should be kept as the same, and the volume of mitochondria is expected to be increased.

MR4: Additive An artificial mitochondrial cluster is added to an image section that would extend an original mitochondrial cluster in adjacent image sections. The artificial

cluster is supposed to be connected to one end of the original one and we expect the original cluster is extended with the artificial one. The total number of the mitochondrial clusters in the reconstructed 3D structure output should be kept as the same, but the volume of mitochondria is expected to be increased.

MR5: Permutative The positions of two mitochondrial clusters are exchanged at the same time in all confocal image sections they appear. During execution, it is necessary to make sure that the exchange will not affect the rebuilding of other mitochondrial clusters. The total number of clusters and volume of the mitochondria in the reconstructed 3D structure output should be kept as the same.

MR6: Invertive Chose one mitochondrial clusters that exists in multiple image section, and then invert the mitochondria in these images (i.e., turn 180^{o} of these images all together). It is important that the inversion will not affect the rebuilding of other mitochondria. The total number of clusters and the volume of the mitochondria in the reconstructed 3D structure output should be kept as the same.

Then, based on the domain knowledge of cell image processing, three additional MRs are developed:

MR7: Lengths Each confocal image section is taken at a different z-axis position of a cell. A small mitochondrion or a small cluster may only appear in one image section due to the gap between adjacent image sections is larger than the size of the mitochondrion, and a large one may appear in multiple adjacent image sections. Based on this observation, if an artificial mitochondrion is added to only one image section, then the 3D structure of the added mitochondrion should be built only based on the added mitochondrion. The newly added mitochondrion is expected to appear in the corresponding processed image section and in the 3D structure output, and the volume of mitochondria is expected to be increased. When the same artificial mitochondrion or cluster is added to two or more adjacent image sections, it is important to check that the 3D structure of the added mitochondrion or cluster can be constructed based on these adjacent image sections. These adjacent sections can be checked by a MATLAB program to ensure the newly added artificial cluster in the image sections will build the same mitochondrion. These image sections and only these image sections should be used to construct the 3D structure of the newly added mitochondrion. The newly added mitochondrion or cluster is expected to appear in the 3D structure output along with the original mitochondria, and the volume of mitochondria will be increased.

MR8: Shapes Mitochondria may have different shapes in the confocal image sections, and the shapes of mitochondria determine the shape of the 3D structure of a reconstructed mitochondrion. Artificial mitochondria with different shapes are added to the confocal images to check whether the 3D structures of these new mitochondria can be constructed as expected, and the 3D structures of other original mitochondria should not be affected.

MR9: Locations The program processes the mitochondria that are close to the nuclear differently to those that are close to the cell membrane. Artificial mitochondria are added to the images in different locations, such as the location where is close to the nuclear or where is close to the cell boundary. The new added mitochondria should be recognized, and the 3D structure of the added mitochondria should be reconstructed as expected and the 3D structures of other original mitochondria should not be affected.

The 9 MRs cover many cases of 3D reconstruction of mitochondria, and they are expected to serve sufficiently as the initial set of MRs. The next step is to create tests based on the identified MRs.

2.2 Metamorphic test generation

Each MT test consists of a source test and its follow-up tests, which are produced through transforming the source test according to an MR. The source tests can be created using regular test generation techniques such as *program-based*, *specification-based*, *combinatorial technique*, or *random* test generation. For each MR, enough tests should be created to adequately cover selected test adequacy criteria. All MT tests are executed one by one, and their outputs are compared according to related MRs to decide whether the SUT passes or fails a test. One way to producing tests is to create tests for each MR independently. For example, for MR1, one can choose a stack of confocal image sections of a cell and then add an artificial mitochondrion in one or multiple image sections. For each test, a different mitochondrion is added, and these mitochondria might be different in size, shape, length, and position. The same idea can be used for producing tests for all other MRs. Another way to producing tests is to use combinatorial technique (Nie and Leung 2011). In this example, two parameters can be defined for the combinatorial test generation. *Parameter1* includes fields {*MR1, MR2, ……, M6*}, and *parameter2* includes fields {*MR7, MR8, MR9*}. Then, pairwise combinations can be applied for producing tests based on the elements of *parameter1* x *parameter2*. For example, tests can be produced based on *MR3* x *MR7,* and tests should be created through resizing mitochondria with different lengths. Tests are created through resizing mitochondria in different positions if the combination is on *MR3* $\times$*MR7.*

2.3 Test adequacy evaluation

When a SUT is executed with tests, the outputs and test adequacy evaluation results are observed and recorded for further analysis. If an output violates an MR, the SUT must have defects. If no MR violation is found, then test adequacy is analyzed to determine whether new tests or even new MRs should be developed. The test adequacy evaluation result is also useful for locating defects. For example, the evaluation result of test criteria could be used for detecting an invalidity of a test input or an enormity of an execution path. Because a test input such as an image could be very large or structurally complex, the SUT cannot fully validate it. In many cases, a test output from an invalid input may still satisfy an MR due the weakness of the MR. The test evaluation results can flag the "false positive" based on unusual coverage information. For example, an output satisfying an MR but with only 1 percent of statement coverage shall flag some problem in the program or the test. In addition, the evaluation of program coverages can improve our understanding of the program execution and detect enormities in execution paths.

Three program coverage criteria are checked when testing the image processing program: *function coverage*, *statement coverage*, and *def-use pair coverage*. In addition, partial mutation testing is conducted. MR- and MR-related tests should adequately cover these criteria and kill or weakly kill all mutants. If a test suite does not adequately cover a criterion, new tests are created guiding by the adequacy evaluation results to improve the test coverage. If 100 percent coverage (or expected threshold) of a criterion cannot be reached through adding new tests, then new MRs that may produce adequate tests should be developed. Code for evaluating the adequacy of coverage criteria is instrumented into the SUT to automate the evaluation at runtime. For *statement coverage*, a checking statement is added immediately before every statement in the program. For *function coverage*, a checking statement is added immediately before every normal return statement

in the function to be checked. For *def-use pair coverage*, one statement is added immediately after the *definition* statement, and the other statement is added immediately after each corresponding *use* statement. For *branch coverage*, checking statement is added immediately before the first statement in every branch that is decided by a decision point. During program execution, results from these checking statements are output to an XML file for runtime and offline analysis.

The effectiveness of MT is measured by the adequacy of program coverage criteria. In addition, mutation testing is also conducted to evaluate the test effectiveness. Ideally, we should create a group of mutants for each MR. However, it is very difficult to know in advance how does a mutant affect an MR in a "non-testable" program due to the equivalent of mutants, which is much worse in MT than regular testing. For example, MR *larger* may not be sensitive to mutants created from changing of constants in a SUT. Therefore, a modified mutation testing is used in this research. In the past few years, significant research has been performed on mutation testing, and many mutation tools have been developed (Jia and Harman 2011). For example, a group of mutation operators were proposed to generate mutants for testing Java programs (Obayashi et al. 1998). When we testing the image processing program, mutants are created manually following general mutation testing practices in addition to domain knowledge. The effectiveness of the testing is evaluated based on the percentage of mutants that are killed. If a mutant was not killed by any of the tests, then new tests or new MRs should be developed until the mutant is killed. Some mutants will never be killed regardless of the tests or MRs. In this case, if a test output cannot be killed by any MR, but the test outputs from the mutated program and the original one are different, then the mutant is counted as weakly killed. If a mutant is weakly killed, inspection is needed to check is there any potential problem in the program. If a mutant cannot be killed or even weakly killed, other testing methods such as the function level MT (Murphy 2010) or analytic methods could be conducted. If all mutants are killed or weakly killed, then it is necessary to check whether these mutants are uniformly killed, which means each MR killed similar number of mutants. If the tests of an MR do not kill any mutant, more mutants should be added to the program to make sure the tests generated from the MR can kill significant number of mutants. Generally, it is difficult to guarantee tests from an MR can kill particular mutants. One example mutant we added for testing the image processing program was applied to the function that is used for smoothing the connection of the contour lines between adjacent image sections in the 3D reconstruction module. If the difference of the mitochondrion shapes between two adjacent image sections is over a limit, the smoothing function is called to smooth the connection lines, which form the boundary of the 3D mitochondrion. Instead, the mutant skips the smoothing function at all. The difference between smoothed connection and the non-smoothed connection may not be easily detected in the 3D structure output by eyes. The number and volume of the mitochondria calculated from the program would not violate any MR. Therefore, the mutant cannot be killed. However, the mutant is easily killed by checking the function and statement coverages due to the inadequacy of function coverage. To simplify the mutation testing, each mutant is tested independently.

It is relatively easy to create mutation tests for evaluating each MR in this case. For example, one can create mutation tests for testing MR1 through adding and removing same number of mitochondria at the same time, and then, we expect the number of mitochondria in the reconstructed structure would not be changed, which violates MR1. It shows that MR1 is a qualified MR for testing the program since they only can be satisfied by carefully selected tests.

2.4 Iterative development of adequate tests and MRs

The major purpose of test adequacy evaluation in the MT is to help testers iteratively develop adequate tests. If adequate tests cannot be developed based on existing MRs, new MRs are needed to be identified for creating more tests. Here, we propose a set of guidelines for iteratively developing adequate tests and MRs guiding by test adequacy evaluation results. We describe the guidelines for each type of criteria separately. It is desirable that the tests of each MR can adequately cover every test coverage criterion. If the tests of an individual MR cannot adequately cover all adequacy criteria, it is acceptable if the tests from all MRs together can do it. We classify the test adequacy criteria into three categories: 1. Program-based criteria such as statement coverage and condition coverage; 2. Mutation testing-based criteria; and 3. MR-related criteria.

1 *Program-based criteria* It is a basic requirement that MR tests can adequately cover control flow-based coverage criteria and data flow-based coverage criteria. Specifically, we discuss the evaluation of evaluate *Statement Coverage, Function Coverage, Branch Coverage and Def-Use Pair Coverage.* If a statement, branch or a function is not covered by any test, some condition in the program is not be covered by any test. However, producing tests for covering all conditions in a program is not an easy work. General test generation strategies such combinational technique, dynamic symbolic execution, category-based test generation, or boundary-based test generation should be used to produce more tests to see whether the new tests can cover the specific conditions. If additional tests cannot improve the coverage, more MRs should be created or some existing MRs should be refined with guiding by the missing conditions in mind.

   Evaluation of the control flow-based coverages offers not only a measurement of the test quality but also a sign of problems in some program sections. In the case study, we will show how the evaluation results help us to find a bug in a program. Checking the adequacy of *def-use* pair coverage is necessary for testing programs that include heavy data processing. The image processing program discussed in this paper is such an example. If the *def* of a *def-use* pair is not covered by any test, but the corresponding *use* is covered, the program must have some problem. If the *def* of a *def-use* pair is covered by a test, but the corresponding *use* is not covered, then it is still the condition coverage problem. The test generation techniques we just talked can be used to create tests and MRs for adequately covering *def-use* pairs.
2 *Mutation Testing* A basic requirement for mutation testing in MT is all tests together should kill or weakly kill all mutants and the test set of each MR should kill or weakly kill some different mutants. A simple way to create a test for killing a mutant is described as follows. First, a pair of MR-related tests such as tc_1 and tc'_1 is selected, and then, tc_1 and tc'_1 is tested under the normal program p. We record the program outputs r_1 and r'_1 of tc_1 and tc'_1 as: $r_1 = p(tc_1)$, $r'_1 = p(tc'_1)$, respectively. Then, run test tc_1 and tc'_1 under the mutated program p_m that has mutant mt, and record the program outputs r_2 and r'_2 of tc_1 and tc'_1 as: $r_2 = p_m(tc_1)$, $r'_2 = p_m(tc'_1)$, respectively. If mutant mt was not killed, both (r_1, r'_1) and (r_2, r'_2) should satisfy their related MR such as mr_1. The differences between r_1 and r_2, r'_1 and r'_2, are represented as $d = r_1 - r_2$ and $d' = r'_1 - r'_2$, respectively, where represents the difference. If (d, d') still satisfy mr_1, then no any test of mr_1 can kill mutant mt since the mutant causes a systematic shifting of the result from the original program. For example, given a mutant that is to change *x2* at line 18 in Fig. 1 into *x1*, then both outputs of test inputs

<3, 2, 5> and <5, 2, 3> from the original program are 3, and the outputs of the same test inputs from the mutated program is 2. The relation (d, d') is (3-2, 3-2), which still satisfies MR permutation and cannot kill the mutant. If the output of a test from the original program and the mutated program is different such as above case, then we considered mutant *mt* be weakly killed. Otherwise, it is necessary to create new tests for killing mutant *mt*. Based on (d, d'), (tc_1, tc_1'), and MR mr_1, in addition to the reference of test coverage results, it is possible to create a new test (tc_{new}, tc_{new}') for killing the mutant. If additional tests from current MRs cannot kill or weakly kill mutant *mt*, new MR and tests should be developed.

3 *Testing with mutation tests* As soon as a SUT passes all tests, a set of mutation tests are created for evaluating the quality of each MR. A set of mutation tests consist of a group of test inputs whose outputs would violate the MR under evaluation, and they are created from mutation of current tests that satisfy the MR. The purpose of testing the SUT with mutation tests is to exclude the MR that is so general that can be hold by all tests and to ensure an MR can differentiate positive test inputs from negative ones. Producing mutation tests for a MR is an iterative experimental process. For each MR, at least one test set that violates the relation should be produced and tested. If an MR satisfies a mutation test, then the MR is too weak or something must be wrong with the MR or the tests.

Based on above discussion, we summarize the MT monitored by test adequacy evaluation as follows:

1. *Development of initial MRs and tests* Based on domain knowledge of the SUT and general framework of MT (Mayer and Guderlei 2006), one can develop a set of MRs. Based on general test generation strategies such as combinatorial technique, category-based or random approach, one produces source tests and follow-up tests for each MR.
2. *Test evaluation* As soon as the SUT passes all tests, tests created for mutations of MRs are used for checking the quality of MRs. The effectiveness of the test is then evaluated with selected test coverage criteria and mutation testing. A mutant should be killed or weakly killed by a test.
3. *Refinement of MRs* It is the process for creating test oracles, which can be developed through refining current MRs or defining new MRs. For example, if simply adding new tests based on current MRs cannot reach 100 percent coverage of the selected test coverage criteria, new MRs should be created or current MRs should be refined until their tests can adequately cover the criteria. If a mutation test satisfies an MR, then the MR should be refined to ensure the MR strong enough to find the violation. Machine learning of testing results and test evaluation data has the great potential for developing adequate MRs (Ding and Zhang 2016).

2.5 Discussion

The advantages of the proposed MT can be summarized as follows: (1) *Using test coverage information for selecting MRs.* The test adequacy evaluation provides a way to measure the quality of selected MRs. It helps testers to develop MRs and determine whether the MRs are adequate for the testing. The test adequacy criteria include program-based coverage criteria, mutation testing, and testing each MR with mutation tests. Mining test evaluation results and testing data for developing high quality MRs is a new direction for MT. (2) *Using test coverage information for generating tests.* Satisfaction of an MR in MT

provides little useful information neither on the SUT nor on the testing process. If a test adequacy criterion cannot be adequately covered by current tests, more tests or even new MRs are needed. The test adequacy evaluation results help testers create specific tests and MRs to improve the coverage. (3) *Test coverage information may flag defects in SUT*. The coverage difference of a group of tests generated from the same MR may flag defects in the SUT. For example, if two tests are produced for the same MR with significant test coverages such as one test covers 90 percent of statements, and the other covers only 45 percent, then the difference may flag some defects in the program or the test inputs.

3 Case study

In this section, we discuss an empirical study of testing a parallel Monte Carlo simulation program developed in Biomedical Laser Laboratory at East Carolina University. The program aims at accurate and efficient modeling of reflectance images from turbid tissue phantoms. More information on the application of the program can be found in a previous publication (Chen et al. 2007). Monte Carlo methods, due to their nature of balancing between algorithm accuracy and computing complexity, have been widely used to simulate light transportation in either homogeneous or heterogeneous turbid media (Kanewala et al. 2015; Le et al. 2014; Pezzè et al. 2007; Shan and Zhu 2009). It involves the stochastic techniques in which random numbers with desired probability distributions are used to model light propagation and interaction in the turbid media that are characterized by the specific optical properties of the system under investigation. The statistical property of the Monte Carlo algorithm enables easy adaptation for problems with irregularly shaped structures and boundaries, which are difficult for numerical approaches. As a general statistical approach, however, the Monte Carlo simulation normally requires tracking a large number of photons (10^7 to 10^9 in this case study) in order to achieve reasonable variance. The simulation consumes a lot of computational time, which tends to be a barrier to practical applications. Fortunately, this problem can be alleviated by a parallel computing algorithm since the tracings of individual photons are independent processes according to radiative transfer theory (Arridge 1999). Adequately, testing the Monte Carlo program is a grand challenge due to the absence of test oracles. The tracking of each photon in the program is controlled by random numbers so that the precise result of each simulation is unknown. The three most widely used approaches, Beer-Lambert law (Arridge 1999; van de Hulst's table Weyuker 1982 and RTE Arridge 1999) can be used as oracles for testing the special cases that simulate photon propagation in homogeneous media. The Monte Carlo program discussed in this case study is for modeling both homogenous and heterogeneous turbid media, where photons propagate in different media such the one illustrated in Fig. 4a. Of course, one can test the program by comparing a Monte Carlo simulation result to an experiment result. However, building an experimental facility as same as the configuration of the Monte Carlo simulation is often expensive and time-consuming, which is the exact reason to build the Monte Carlo simulation program. This is a common character of many scientific software: using the simulation software to replace the physical experiment study. Traditional software testing techniques cannot adequately test the Monte Carlo program. Therefore, MT monitored by test adequacy is used to test the program.

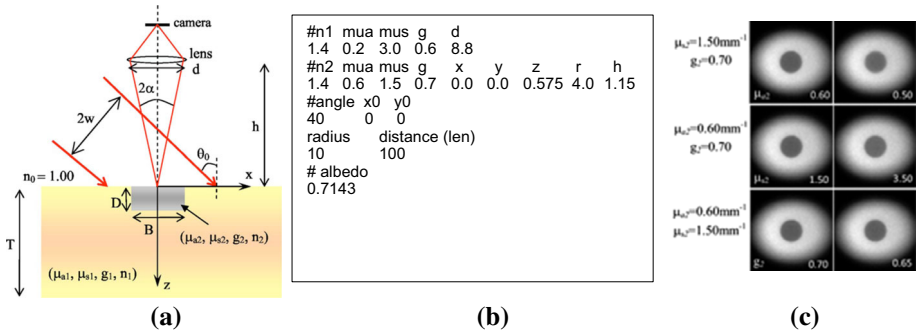

Fig. 4 **a** Configuration of the simulation for acquisition of spatially resolved images (Chen et al. 2007); **b** Partial of the input file of the simulation program; and **c** 6 simulated reflectance images obtained with different optical parameters

3.1 Monte Carlo modeling of reflectance imaging

The Monte Carlo modeling program in this case study is built for numerical investigation of the dependence of reflectance image data on the optical and geometric parameters of heterogeneous tissue phantoms under the condition of full-field illumination. To simulate the light distribution reflected from a heterogeneous tissue phantom, different regions in the phantom are assigned with different sets of optical parameters as shown in Fig. 4a. In Fig. 4a, the diameter of the incident light beam is $2w$, the incident light angle is θ_0, h is the height of the camera lens, d is the diameter of the camera lens, 2α defines the Field Of View (FOV), which defines the angular range of reflected photons to be captured by the camera lens, T is the thickness of the host phantom, D and B are the thickness and diameter of the cylindrically shaped and embedded phantom, respectively. (μ_{a1}, μ_{s1}, g_1, n_1) and (μ_{a2}, μ_{s2}, g_2, n_2) are the optical parameters to characterize, respectively, the two phantom regions: the absorption coefficient, scattering coefficient, anisotropy factor, and refractive index. For semi-infinite (only considering $z \geq 0$) homogeneous phantoms, we let $T \rightarrow \infty$ and (in simulations, $T = 100$ *mm*) $D = B$=0 *mm* for the insert (Chen et al. 2007). The Monte Carlo simulation program considers a homogeneous phantom or a heterogeneous phantom in air with one embedded region of optical parameters different from its semi-infinite host. The independent tracking of photons makes Monte Carlo simulations ideal for parallel computing. The program employed Message Passing Interface (MPI) for parallelization of the sequential Monte Carlo code. The random numbers used in the Monte Carlo simulation are generated independently and uniformly between 0 and 1 for describing the random events of light scattering and absorption. Hundred or more random numbers per tracked photon in most cases here are required in a simulation. The random events in the photon tracking process are sorted into different types according to the nature of the light tissue interaction such as the scattering, absorption, and reflection from or refraction through an interface. To ensure the randomness for accurate Monte Carlo modeling on the basis of ergodic hypothesis, it is necessary to assign each type of random events a unique sequence of random numbers. Therefore, several independent random number sequences coexist in a Monte Carlo simulation (Chen et al. 2007).

Just like other scientific software, it is difficult to adequately test the Monte Carlo program since one cannot predict the result of a simulation with an input configuration similar to the one shown in Fig. 4b. In fact, that is the knowledge we try to learn from the

modeling. According to the discussion above, it is also very hard to use other algorithms to yield suitable test oracles and indicate the expected output for a certain input. It is either very time-consuming or even worse in most cases—no alternative algorithm is available to generate reliable test oracles for the modeling, especially, for the system with irregular-shaped elements or complex boundary conditions, which probably happens all the time while modeling the real-world problem. An output of the Monte Carlo program is a reflectance image as those shown in Fig. 4c.

3.2 Program structure

The Monte Carlo simulation program was developed using Fortran 90 with an MPI library (Pacheco 1996). The program contains five source files. *Monte_main.f90* is the *main* program including the code calling the MPI functions; *Monte_go.f90* includes the subroutines and functions to check if a photon hits different optical boundaries in the turbid medium and record current photon status and position. *Monte_sub.f90* is the module of utility subroutines that do the calculation in the simulation. *Monte_io.f90* is the file for input/output subroutines; *Monte_define.f90* contains definitions of objects and constants. The program has 46 subroutines or functions, and total about 1600 lines of Fortran 90 code, which does not include third party software like random generators.

3.3 Experiment setup

Each test was carried out on a computing cluster of 4 nodes (PowerEdge 1750, Dell) with a total of 8 process elements (Xeon 3.06 GHZ CPU). Intel MPI library was used as the message passing interface. Two structural test coverage criteria, *function coverage* and *branch coverage*, were checked for all subroutines and modules in the program. In order to reduce the variance in the simulation, averaged reflectance curve $R(x,0)$ was calculated by the photon density along the y-axis over three rows of grid cells on each side of the x-axis. The contrast C of a reflectance image is calculated based on the intensity (*i.e.*, the unsigned value of a pixel) of each pixel in the image. The MRs selected in this research originated from the results validated in (Chen et al. 2007). In the Monte Carlo program, different configurations of the simulation are carefully set as the test inputs and the relations among the outputs of those configurations are established according to optical knowledge. We assumed the relations established in (Chen et al. 2007) are valid so that we can conduct MT, and the initial MRs were developed based on the relations.

3.4 Test execution

3.4.1 Identification of MRs

We first identify a set of MRs based on domain knowledge and general guidelines. The input parameters to the Monte Carlo program are the simulation configuration like the one shown in Fig. 4b and the configuration of incident light beam, which includes thousands of datum items produced by experiments. We do not change the light incident beam configuration except its angle θ_0. When the light beam angle is changed, the corresponding light configuration can be simply calculated based on the original light configuration. Therefore, we do not test the program with different incident light beams except different light beam angles. The output of a simulation is the number of backscattered photons

divided by that of the incident photons as the intensity of each pixel within the FOV captured by the camera lens, which produces a gray level image. It is infeasible to define an MR for an input parameter and the individual pixel values of the output reflectance image. Instead, we define MRs for the input parameters and selected properties of the output image. The input parameters we consider first include refractive index n, anisotropy factor g, numerical aperture NA ($NA = sin\alpha$, where α is the angel of FOV), incident light beam angle θ_0, and albedo α, which given by $\mu_s/(\mu_a + \mu_s)$). The input parameters cover all major parameters in the simulation configuration. The correlation between the parameters and the simulated reflectance image is also the purpose for building the program. Since the phantom in the simulation configuration is the object we investigate in the simulation, parameters of the media around the phantom will not be considered. The properties of the output image include the image contrast C and reflectance curve $R(x, 0)$, both of them are defined in Sect. 3.3. The ideal scenario is that we create MRs based on combinations of input and output parameters. However, we only know some of the relations based on theoretical and experiment results. Five MRs are first defined based on the experiment results that have been published (Chen et al. 2007). The five MRs are defined as follows:

1. MR1: Contrast C value decreases when refractive index n_2 value increases.
2. MR2: Contrast C value decreases when anisotropy factor g_2 value increases.
3. MR3: Contrast C value increases when albedo α_2 value increases.
4. MR4: For each pixel along the x-axis on the image, the averaged reflectance curve $R(x, 0)$ decreases if the numerical aperture (NA) decreases.
5. MR5: For each pixel along the x-axis on the image, the averaged reflectance curve $R(x, 0)$ decreases if the incident light angle θ_0 increases.

These properties are based on experimental results, and they were not defined in the program specification or explicitly implemented in the Monte Carlo simulation program. In this perspective, MT the program with these properties is conducted for validation of the program (Zhou et al. 2009).

3.4.2 Test generation

For each MR, at least two test sets with different optical parameters were crossly validated to lower the possibility that the MR was satisfied accidentally by special inputs. Cross-validation alone is not enough to ensure the testing quality, instead additional test adequacy evaluation such as mutation testing is needed (Xie et al. 2011). Each test set contained at least four tests that satisfy the MR. These tests include boundary values that are uniformly distributed within the valid value range of the parameter. For example, the 4 values selected for test set 1 of MR1 is 1.36, 1.40, 1.44 and 1.48 with the valid value range between 1.0 and 1.5 of reflective index n_2. The test results of the second test set are not presented in this paper as long as the test results are consistent to the one of the first test set. For the same reason, the detailed configuration inputs are skipped except tests for MR1. We only describe the parameters that are needed for the testing.

3.4.3 Testing results

(1) *MR1 Contrast C value decreases when refractive index n2 value increases.*

Test Set 1: $N_0 = 1.13*10^8$, light incident angle $\theta_0 = 30.0°$, height of collection lens = 0.0 mm. Phantom: $\mu_{a1} = 0.30\ mm^{-1}$, $\mu_{s1} = 5.50\ mm^{-1}$, $g_1 = 0.8$, $n_1 = 1.40$,

$T = 100$ mm. Embedded Cylinder: $\mu_{a2} = 0.15$ mm^{-1}, $\mu_{s2} = 6.00$ mm^{-1}, $g_2 = 0.8$, $D = 0.75$ mm, $B = 8$ mm, Cylinder center position $(x, y, z) = (0.0$ mm, 0.0 mm, 0.375 mm). The n_2 of the four tests are set as 1.36, 1.40, 1.44, and 1.48. The simulation results and test adequacy evaluation results are shown in Table 1.

Fun-go, *Fun-sub,* and *Fun-io* are the coverage of all functions or routines in module *monte_go*, *monte_sub,* and *monte_io,* respectively. *Branch-go*, *Branch-sub,* and *Branch-io* is the coverage of all branches in module *monte_go*, *monte_sub,* and *monte_io*, respectively. The results of contrast *C versus* n_2 are plotted as shown in Fig. 5. The figure was visually inspected to decide whether the test results satisfy MR1. As the conclusion, the test results of test set 1 satisfy MR1. If we add test inputs to cover the exception handling code, then all branches will be covered. Tests for covering exception handling were not considered for MR1, same for other MRs.

(2) *MR2 Contrast C value decreases when anisotropy factor* g_2 *value increases.*

The values of g_2 of the four tests are set as 0.1, 0.3, 0.6, and 0.9. The results of contrast *C versus* g_2 are plotted as shown in Fig. 6, and the test results satisfy MR2. All branches except the exception handling and all functions are covered by the tests.

(3) *MR3 Contrast C value increases when albedo* α_2 *value increases.*

The values of μ_{a2} of the four tests are set as 0.2, 1.00, 2.5, and 5.00 mm^{-1}, respectively. All branches except the exception handling code and all functions are covered by the tests. The results of contrast *C versus* α_2 are plotted as shown in Fig. 7, and the test results satisfy MR3.

(4) *MR4 For each pixel along the x-axis on the image, the averaged reflectance curve R(x,0) decreases if the numerical aperture (NA) decreases.*

The height of collection lens of the four tests are set as 0.00, 3.35, 12.50, and 46.65 mm, which make the numerical aperture (NA) as 1.000, 0.966, 0.707, and 0.259, respectively. The simulation results and test adequacy evaluation are shown in Table 2. As shown in Table 2, some of the branches were not covered by the tests. But after we added new test with the lens height as 50.00 mm, all branches except the exception handling code and all functions are covered by the tests. The results of averaged curves of *R(x,0) versus x* are plotted as shown in Fig. 8, and we can conclude that the test results satisfy MR4 via visually inspecting all Figures

5) *MR5 For each pixel along the x-axis on the image, the averaged reflectance curve R(x,0) decreases if the incident light angle* θ_0 *increases.*

The incident light angles θ_0 of the four tests are set as 0, 15, 45, 75°, respectively. All branches except the exception handling code and all functions are covered by the tests. The

Table 1 Testing results of MR1

n_2	Contrast *C*	Fun-go (%)	Fun-sub (%)	Fun-io (%)	Branch-go (%)	Branch-sub (%)	Branch-io (%)
1.36	0.19589	100	100	100	75.9	60.7	90
1.40	0.175588	100	100	100	74.7	60.7	86.7
1.44	0.162571	100	100	100	76.5	60.7	86.7
1.48	0.148713	100	100	100	76.5	60.7	86.7

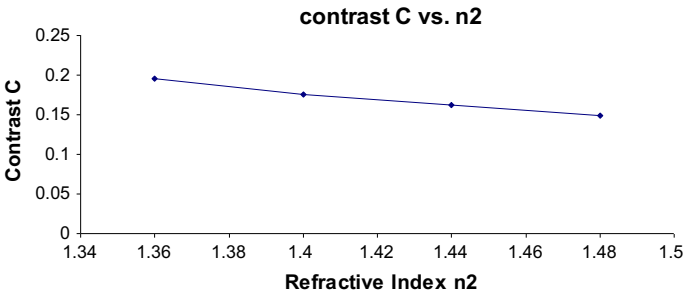

Fig. 5 Contrast *C* versus refractive index *n2*

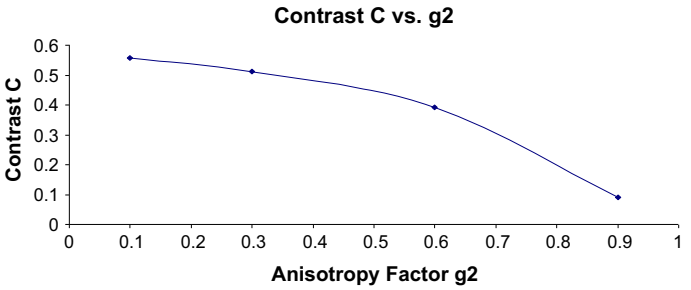

Fig. 6 Contrast *C* versus anisotropy factor g_2

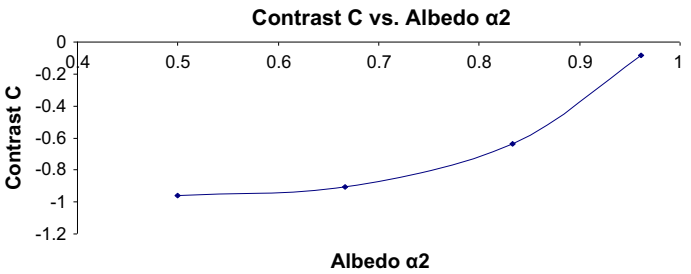

Fig. 7 Contrast *C* versus albedo α_2 value

Table 2 Testing results of MR4

h(mm)	NA	Fun-go (%)	Fun-sub (%)	Fun-io (%)	Branch-go (%)	Branch-sub (%)	Branch-io (%)
0.00	1.000	100	100	100	73.5	60.7	86.7
3.35	0.966	100	100	100	73.5	60.7	86.7
12.50	0.707	100	100	100	73.5	60.7	86.7
46.65	0.259	100	100	100	73.5	60.7	86.7

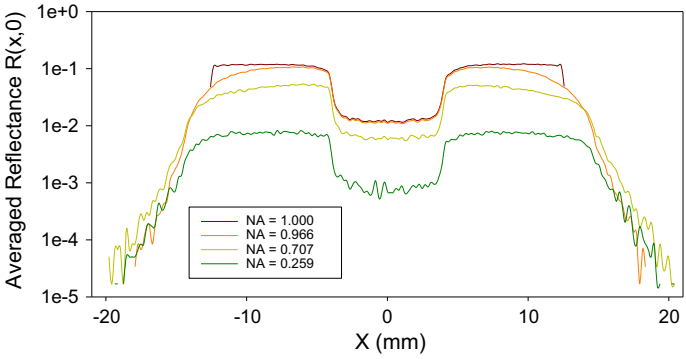

Fig. 8 Average reflectance *R(x,0)* versus *x*

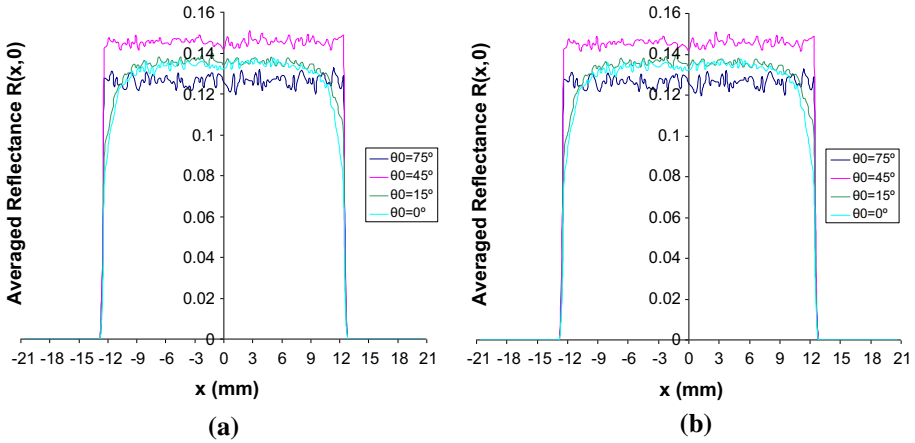

Fig. 9 **a** Average reflectance $R(x, 0)$ *versus* x with different incident light angle θ_0 in heterogeneous media. The result violates MR_5. **b** Average reflectance $R(x,0)$ *versus* x with different incident light angle θ_0 in homogenous media. The result still violates MR_5

results of *R(x,0) versus x* are plotted as shown in Fig. 9a, and the figure was visually inspected to decide whether the MR is satisfied. We found there was no simple linear relation between averaged reflectance curve $R(x,0)$ and x, which violates MR5 as a property reported in reference (Chen et al. 2007). The value of $R(x,0)$ at $\theta_0 = 45^{\circ}$ is larger than the one at $\theta_0 = 75$, 15, and 0°. Based on MR5, $R(x,0)$ at $\theta_0 = 45°$ shall larger than the one at $\theta_0 = 75°$ but smaller than the one at $\theta_0 = 15$ and 0°.

In order to investigate the problem, we cross-validated several different sets of tests but could not find any test result that satisfied MR5. However, MR5 could be satisfied with some special tests, such as $\theta_0 = 45, 75°$ in above test set. The result further confirms that it is important to evaluate the quality of the tests and their corresponding MRs in order to minimize the false positive results. From physics knowledge and the configuration of the simulation, one knows that if MR5 is satisfied in heterogeneous media, then it should be satisfied in homogenous media as well. Since the code for handling the homogenous media in the program is a subset of the code for handling the heterogeneous media, we tested MR5 for homogeneous media to check whether MR5 is satisfied in this special case.

The parameters of the embedded cylinder of all tests in the test set were selected as $D = 0.0$ mm, $B = 0.0$ mm, cylinder center position $(x, y, z) = (0.0$ mm, 0.0 mm, 0.0 mm), and the optical parameters of the cylinder were set as same as the optical parameters of the phantom. The incident light angles θ_0 of four tests were still set as 0, 15, 45, and 75°. The test results and test adequacy evaluation are shown in Table 3. The results of $R(x, 0)$ *versus* x are plotted as shown in Fig. 9b, and it was visually inspected. It is easy to see that no simple linear relation between averaged reflectance $R(x,0)$, and x is revealed for homogeneous media either.

Checking the test coverage results in Table 3, it is not difficult to find that the program for handling the homogenous media is much simpler than the program for handling the heterogeneous media. Since Fig. 9a, b have the same pattern, we can narrow the defect code to the program for processing homogenous media. Several experienced programmers independently inspected the program for homogenous media (less than 500 lines of code), and no bug was found. From theory, it is difficult to tell whether MR5 holds or not. One way to check the validity of MR5 is to build a real experiment environment to test MR5. However, building the experimental environment that is identical to the configuration of the Monte Carlo simulation is fairly hard. The scientists who developed the Monte Carlo model believed MR5 should be correct. Therefore, further investigation is necessary. Based on the testing results of MR5, we found one important observation: MR5 helps us narrow the defect scope in the program. For example, the defects should exist in the program for processing homogenous media because the testing results from program for processing homogenous media and heterogeneous media have the same pattern. The observation actually helped us find the defect in the program.

Carefully checking the result shown in Fig. 9a, b, it is not difficult to find that the phantom defined in the simulation configuration is not shown. Based on the configuration, the figure should have a valley in the middle representing the phantom defined in the middle of the configuration. This suggests errors within the program. By analyzing the program, we found that one statement in the program was hard coded with $\theta_0 = 40°$ instead of reading the value from an input file (but reading θ_0 in other places is correct). As soon as the problem is fixed, the running results of the same test inputs are plotted as shown in Fig. 10, which satisfies relation MR5. The experiment has again shown the effectiveness of MT and the importance of MRs for detecting defects. If MR5 were not selected as an MR for the testing, the defects discovered by MR5 would not be found.

3.5 Test evaluation

The test results in Sect. 3.4.3 show that the test covered all functions, conditions and statements except exception handling. In this section, we discuss test evaluation with mutation testing and testing MR with mutation tests.

Table 3 Testing results for MR5 (homogeneous)

θ_0 (°)	Fun-go (%)	Fun-sub (%)	Fun-io (%)	Branch-go (%)	Branch-sub (%)	Branch-io (%)
0	78.6	100	100	33.7	62.5	86.7
15	78.6	100	100	33.1	57.1	86.7
45	78.6	100	100	33.1	57.1	86.7
75	78.6	100	100	33.1	57.1	86.7

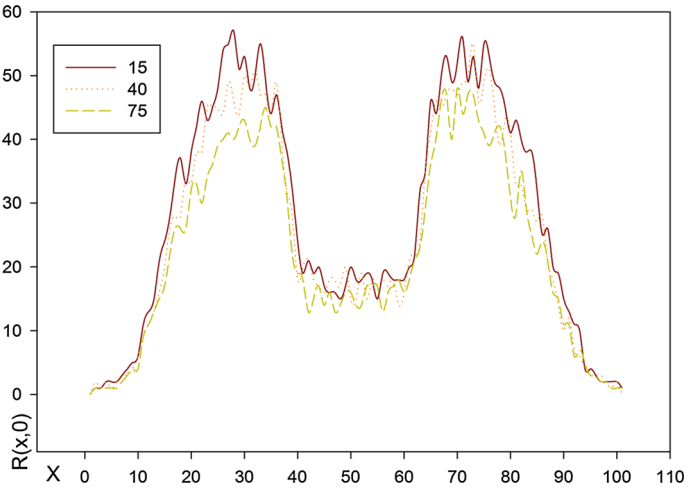

Fig. 10 Average reflectance curve $R(x,0)$ versus x. The result satisfies relation MR_5 and it correctly shows the phantom defined in the simulation

3.5.1 Mutation testing

The test sets used for checking the mutation testing are those generated from MR1 to MR5. Mutation operators were created for modifying the calculation in the Monte Carlo simulation program, and all mutation operators were only applied to the module *monte_go.f90* and the module *monte_sub.f90* since they are the main computing modules of the simulation. No mutant affects the validity of tests, and no mutation was applied to the exception handling code or input and output statements. In addition, no mutant causes any simple exception or crash during the testing. Mutants were manually instrumented into the program, and each mutant was checked separately. Table 4 lists the mutation operators that were applied to the Monte Carlo program.

There are 11 subroutines and functions in module *monte_go.f90* and 13 subroutines and functions in module *monte_sub.f90*. We created at least three different types of mutants for each function or subroutine except two subroutines that only have several lines of simple statements. Total 150 mutants were created, and each of them was instrumented and tested separately using the same tests for testing MR1 to MR5. We restricted the value modification of each coefficients or constant less than 10 % of the original value, and each constant added to a formula was less than 10 % of the expected correct result. Float numbers are rounded to integers if needed. Since 0 or 0.0 normally is very sensitive to any change, no mutation was applied to them. The results generated from each mutation program were compared to the results generated from the original program and checked manually. Majority of mutants were killed, and all of them were weakly killed. We found the Monte Carlo program is very sensitive to mutants. A simple change could cause a catastrophic error, which might be a common characteristic of scientific software since the computation in this type of software is precisely set, and it cannot tolerate any change. Table 5 shows the mutation testing results.

For mutant type 1 and 2, the mutation scores could not be improved even when new tests were added to the original tests. However, when the mutation operator 1 or 2 was

Table 4 Mutation Operators

Operators	Description
1. Modify the coefficients in a formula	Change the coefficients in a formula for simulating the photon propagation.
2. Add a constant in a formula	Add a negative or positive value to a formula.
3. Change an operator in a formula	Change an operator in a formula.
4. Change an operator in a conditional statement	Change an equality or relational operator in a conditional statement.
5. Change a constant in a conditional statement	Increase or decrease the value of a constant in a conditional statement.
6. Remove a clause in a conditional statement	Remove a sub-condition in a conditional statement.
7. Remove a case handling	Remove one case handling in a multiple case handling section.
8. Modify the photon moving direction	Assign a photon a different direction to go.
9. Modify the photon hitting types	Assign a photon a different type of media it hits.

Table 5 Mutation testing results

Mutant Types	Number of Total	Number of Dead	Dead (%)	Killed by MRs	Comment
1	30	26	87	1, 2, 3, 4, 5	The four mutants that were applied to the formula for calculating the number of photons could not be killed.
2	15	4	27	1, 2, 3, 4, 5	The four mutants that were killed were those applied to the formula for detecting the boundary of photons in the media.
3	30	30	100	1, 2, 3, 4, 5	Four of them were killed by causing exceptions.
4	30	30	100	1, 2, 3, 4, 5	Six of them were also flagged by the low percentage of branch coverage.
5	15	15	100	1, 2, 3, 4, 5	All of those constants represent media types or photon prorogation directions.
6	15	15	100	1, 2, 3, 4, 5	Example: *Dir(3) < 0.0.AND. Pos(3) > = Reg(2)*, then remove (*Dir(3) < 0.0.AND.*) or (*.AND. Pos(3) > = Reg*(2)).
7	5	5	100	1, 2, 3, 4, 5	Mutation operators were only applied to the cases for complex calculation.
8	5	5	100	1, 2, 3, 4, 5	All of them were also flagged by the low percentage of branch coverage.
9	5	5	100	1, 2, 3, 4, 5	All of them were also flagged by the low percentage of branch coverage.
Total	150	135	90	1, 2, 3, 4, 5	Cannot kill all mutants.

applied to different formulas, the mutation scores were different. The mutation scores indicate that the MT monitored by test adequacy is an effective technique for testing the 'non-testable" Monte Carlo simulation program.

3.5.2 Testing MRs with mutation tests

Generating mutation tests is tricky since one expects any outputs from valid test inputs should not violate any MR. For example, the contrasts of two images calculated by the Monte Carlo simulation program with any valid pair of refractive index n_2 should satisfy relation MR1, assume values of other parameters of the two simulations remain the same. It is impossible to create a set of mutation test inputs for MR1 by just adjusting only the value of n_2. The values of other parameters in the configuration file have to be changed in order to produce outputs that will violate MR1. Due to the complex correlation among the parameters, tools such as MATLAB are needed for producing the mutation tests. As soon as all MRs have passed the test, we use MATLAB to visualize the testing results like those shown in Fig. 7 (for MR3), check the satisfaction of each relation, and then find mutation test inputs based on the visualized results. When we test MR3, all tests have same values for all parameters except albedo α_2. In order to create a mutation test for MR3, not only the change of albedo α_2 is made, but also the change of index value is made such as in one test, n_2 is set as 1.4, and another one n_2 is set as 1.5 (in normal test, n_2 should be set as the same value for MR3). Then, relation MR3 will not be held by the two tests due to the change of both index and albedo values together, as shown by the red line in Fig. 11, where *the contrast value decreases when the albedo value increase*. Using the same idea, we created two mutation tests for each MR, and each test produced a result that violated the MR under test. It shows that MR1 to MR5 are well designed, and they are only satisfied by selected tests. Therefore, they are effective for testing the program.

4 Discussion

We developed five MRs for MT of the Monte Carlo simulation program. While some of the MRs can be validated by physics theory or experimental results, others (such as MR5) are difficult to be validated. We selected these MRs based on experiment results (Chen et al. 2007). Several test suites were created for each MR, and each test suite includes at least four tests. For each test (except those selected for MR5 for homogeneous media), all

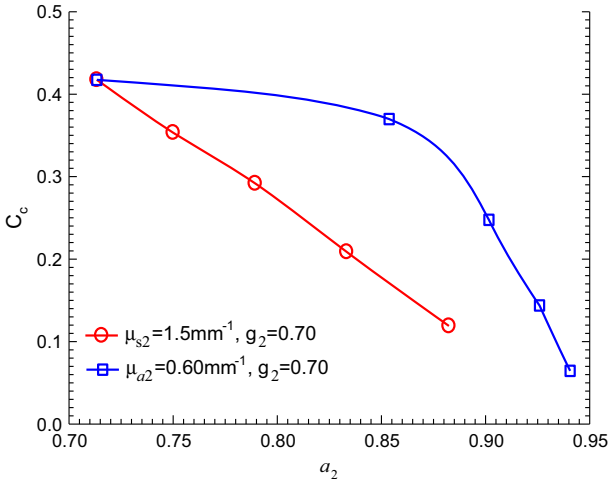

Fig. 11 *Red line* results are produced with mutation tests and violates relation MR3 (Color figure online)

functions and all branches except those handling exceptions were covered. In order to cover all branches, new tests that trigger the exception handling should be added. The results of MT on the Monte Carlo program show that test coverage results can be used for evaluating the quality of tests and MRs, and the results can be used as a guideline for selecting adequate MRs and creating tests. The same tests for testing MR5 in heterogeneous media have much higher branch coverage than when they are used for testing MR5 in homogeneous media. The coverage difference is useful for narrowing the problem when testing MR5. If complex test coverage criteria such as state transition coverage or path coverage are considered, the coverage results can be more useful in evaluating the quality of MRs and their tests. The coverage results of all of those tests are very similar, which indicates that simply increasing the number of tests for each MR is of little help for finding more defects. The test results of MR5 showed that some defects must exist in the Monte Carlo simulation program. Through testing MR5 for both homogenous and heterogeneous media, the testing results showed that we can narrow the scope of the potential problem in the program via analyzing the test coverage results to reveal possible defects, which eventually were found and corrected. The mutation testing results further showed the effectiveness of MT monitored by test adequacy, and they suggest the need of additional MRs to kill all mutants. The testing MRs with mutation tests showed that the selected 5 MRs were effective for testing the Monte Carlo program. In brief, the case study has demonstrated that the extension of MT with test adequacy evaluation can provide useful information for developing better tests and MRs for testing complex "non-testable" software.

5 Related work

In software testing, the pass/fail verdict of tests depends on the availability of test oracles, which define the comparison between test outputs and their expected results (Barr et al. 2015; Baresi and Young 2001). Some software systems are "non-testable" due to the absence of test oracles. The focus of this research is on adequately testing "non-testable" software particularly on scientific software. Many scientific software systems are belonged to "non-testable" software, which is often tested with pseudo oracles that are derived from a different implementation, solutions obtained analytically, simplified data, references from experiment and theoretical study, MT and many other approaches (Kanewala and Bieman 2014). But existing test approaches are not adequate for assuring the quality of scientific software (Kanewala and Bieman 2014; Weyuker 1982). For example, Beer-Lambert law, van de Hulst's table, or Radiative Transfer Equation (RTE) can be used for testing a Monte Carlo simulation program that is used for modeling light propagation (Arridge 1999; Chen et al. 2007; Keijzer et al. 1989), but they are only applicable to the system that models homogeneous media (Chen et al. 2007). Therefore, these methods can be looked at as testing with special cases, which is not sufficient to ensure the quality of the software. Adequately testing scientific software is a grand challenge. Kanewal and Bieman classified the challenges into two categories: the challenges caused by the characteristics of scientific software, and the challenges caused by the cultural differences between scientists and testers (Kanewala and Bieman 2014). The first challenge that occurs due to the characteristics of scientific software is the oracle problem. The key for adequately testing scientific software is to solving oracle problems (Kanewala and Bieman 2014). MT has demonstrated its effectiveness in addressing oracle problems through successfully testing

many "non-testable" programs such as bioinformatics systems (Chen et al. 2009), machine learning systems (Xie et al. 2011), compilers (Le et al. 2014), scientific computing systems (Chen et al. 2002; Ding and Zhang 2016), large-scale database (Lindvall et al. 2015), and online service systems (Chan et al. 2007; Zhou et al. 2015). A controlled experiment of three open source programs by 38 testers has shown that MT is cost-effective in fault detection (Hu et al. 2006), and the result of an empirical study has summarized the fault detection capability of MRs (Liu et al. 2014). A recent application of MT to the complier testing has found over 300 bugs in several widely used C/C++ compilers (Le et al. 2014). MT was also successfully used for the assessment of the quality of search engines Google, Bing and Baidu (Zhou et al. 2015).

The most important tasks in MT are the identification of adequate MRs and generation of adequate tests. Several methods on the development of MRs have been proposed. Murphy, Kaiser, Hu and Wu classified six types of MRs for testing machine learning systems (Murphy et al. 2008). They include *additive* or *multiplicative*, which increases or decreases or multiplies a source test by a constant value to produce a follow-up test, and the source test is paired with the follow-up test to test the SUT according to *additive* or *multiplicative* relation; *permutative,* or *invertive*, which permutes or reverses the order of data elements in a source test to produce a paired follow-up test; *inclusive* or *exclusive*, which adds or removes some new datum items in a source test to produce a paired follow-up test. The six MRs have been widely used for guiding the development of MRs, and they were extended in later work (Xie et al. 2011). The MRs used for testing the 3D structure reconstruction program in this paper used the similar MRs. However, the six MRs are restively weak so that they are not enough to test a complex program, and the effectiveness has to be rigorously evaluated. In this paper, we evaluated each MR with program-based coverage, mutation testing, and testing MRs with mutation tests. Murphy also proposed a set of general guidelines for identifying metamorphic properties according to business rules, algorithms used in the applications, the implementation of the algorithm, and special inputs of the application (Murphy 2010). The guidelines offer a good direction for identifying MRs, and its natural extension is to define a set of criteria for evaluating MRs and refining MRs. Chen, Poon and Xie proposed a systematic methodology for identifying MRs based on the combination of categories of test input spaces (Chen et al. 2015). In the framework, the input space of the SUT is divided into several categories, and then MRs are developed based on the combination of the categories. The idea behind the approach is to create MRs that can adequately cover the SUT. The method is useful for developing adequate MRs and tests for testing scientific software such as the Monte Carlo simulation program. Murphy, Shen and Kaiser studied function level metamorphic testing, where the specifications of metamorphic properties are transformed into runtime assertions in functions to ensure that the specification hold during program execution (Murphy et al. 2009; Murphy 2010). Comparing to system level MRs, function level MRs are easily to be precisely defined. Therefore, function level MRs can detect even more defects in "non-testable" programs than system level MRs (Segura et al. 2016), which is due to the weakness of system level MRs. However, in order to adequately test a program using MT, system level MRs are necessary. How to combine function level MRs into system level MRs should be an interesting direction for MT. MRs also can be developed through program analysis and machine learning. An integrated testing approach based on symbolic execution and MT has been proposed (Chen et al. 2002). In the approach, all possible outputs under path conditions of a symbolic input and all symbolic inputs generated according to an MR are produced by the symbolic execution, and then an input and its corresponding output is checked for the violation of the MR under every path condition

(Chen et al. 2002). Knewala, Bieman, and Ben-Hur recently reported a result on the development of MRs for testing scientific software through machine learning data flow and control flow information extracted from SUT (Kanewala et al. 2015). Its effectiveness has been demonstrated through discovering the seeded bugs in several experimental programs. However, these MRs are very general and the programs are tiny. These MRs could be easily developed based on general MR development guidelines and domain knowledge. The applicability of the approach to large programs is unclear. However, building MRs iteratively through machine learning testing data and testing evaluation results could be a potential solution to the oracle problem (Ding and Zhang 2016).

Test generation is another important task in MT. Traditional test generation techniques such as program-based, specification-based, and parameter-based test generation are important for producing source tests in MT. The source tests and their MRs-related tests are used together for testing SUT in MT. Obviously, the quality of the source tests directly affects the coverage of test adequacy criteria. Application of specification-based test generation to scientific software like Monte Carlo simulation could be effective since the specification of this types of programs normally are much formal and it is much more abstract than the implementation. Therefore, it is relatively easy to produce adequate source tests. For example, source tests of the Monte Carlo program can be automatically produced with a Monte Carlo formal model. However, the method cannot solve the oracle problem since the correct output of an individual test is still unknown. Gotleib and Botella have reported how to translate an SUT into an equivalent constraint logic program and convert a fault-based model of an MR into a goal to be solved in the constraint logic program (Gotleib and Botella 2003; Gotlieb et al. 1998). The solution of the goal, if found, is corresponding to a set of tests for revealing potential violations of the MR in the SUT (Gotleib and Botella 2003). However, due to the complexity of the constraints to be solved, the applicability of the approach is limited. It is essentially a program analysis technique. Parameters-based test generation produces tests based on combination of input parameters such combinatorial technique, category-based technique and random generation. All of these techniques have been widely used for producing source tests in MT (Chen et al. 2015; Ding and Zhang 2016; Mayer and Guderlei 2006). Murphy introduced an automated MT framework, where the SUT runs with both a source test and an MR-related follow-up test together (Murphy et al. 2009), and the specification of the MR is provided to the program so that the outputs can be automatically checked. They also discussed heuristics for reducing false positives and dealing with non-deterministic test outputs through thresholds and ranges of acceptable outputs (Murphy et al. 2009). Mayer and Guderlei proposed a random test generation approach for testing image processing applications (Mayer and Guderlei 2006). A new image is generated through randomly selecting each black pixel with a probability from a reference image, and then the image is transformed into another image based on an MR such as rotation or intersection of images. The translated image becomes a test input for testing the image processing application, and its output can be verified according to the MR that is used for producing the image. The same idea was used in this paper, where a confocal image was manipulated with mitochondria for producing new images. Shan and Zhu reported a method for producing structurally complex tests through data mutation (Shan and Zhu 2009). A set of mutation operators are defined and a set of seed tests are created, and then a set of new tests are generated through applying mutation operators to the seed tests (Shan and Zhu 2009). We used the approach for creating mutation tests for testing MRs. Guderlei and Mayer proposed a statistical MT, where two or more output sequences are generated and compared according to the statistical MRs defined on the program outputs (Guderlei and Mayer 2007). The test

generation techniques for producing source tests and MR-related follow-up tests are important for producing adequate tests. Our research focus is on evaluation of the adequacy of the tests and iterative generation of tests through analyzing testing data and test evaluation results, which were rarely systematically discussed in existing MT publications.

The evaluation of MT is focused on the selection of MRs since the effectiveness of MT is highly depended on the selection of MRs. Many researchers support to conduct MT using as more as possible MRs (Zhou et al. 2009). However, many redundant MRs could be developed, and they could not improve the testing effectiveness. Therefore, rigorously evaluating MRs with test adequacy criteria is important for selecting good MRs. Chen, Huang, Tse, and Zhou discussed an approach for selecting MRs that could be good at fault detection (Chen et al. 2004). Their case studies indicate that domain knowledge alone is inadequate for finding good MRs. Good MRs should be selected considering the implementation of the SUT (Chen et al. 2004). However, Mayer and Guderlei had different conclusion. They claimed that good MRs should be those defined based on the semantics of the SUT (Mayer and Guderlei 2006). Asrafi, Liu, and Kuo pointed out good MRs are those that can make the multiple executions of the program as different as possible (Asrafi et al. 2011). The results further confirm the importance of the evaluation of MRs with program-based coverage criteria. In addition, mutation testing and testing MRs with mutation tests are also important to evaluate the effectiveness of MRs. There are few work on mutation testing of MT such as mutation testing was used for evaluating MT for testing machine learning algorithms (Zhou et al. 2009), but we have not seen any other work on evaluating MRs with mutation tests. Our research results could provide a reference for rigorously evaluating MRs. More important, we have shown the evaluation results can be used for iteratively developing adequate MRs.

6 Summary and future work

Metamorphic testing is an effective technique for testing systems that do not have test oracles. Checking outputs against MRs alone is inadequate for ensuring the quality of the SUT. In this paper, MT is enhanced by the evaluation of test adequacy criteria: *program-based test coverage criteria, mutation testing, and testing MRs with mutation tests*. The evaluation of test adequacy criteria should be a requirement of MT. It serves as a guideline for selecting MRs, generating tests, and finding defects in the SUT. The effectiveness of our approach has been demonstrated by testing a image processing program and a Monte Carlo simulation program, both are "non-testable" scientific software systems. In addition to checking five MRs, the test adequacy is evaluated during the testing process for refining and creating MRs and tests. The proposed testing approach with example and case study results would be useful for testing other scientific software. The case study also raised several questions regarding the quality of MT. For example, how can we construct adequate MRs in general? The MT monitored by test adequacy provides a primitive solution to solve the issue. In the future, we will investigate constructing MRs through machine learning software engineering repositories.

Acknowledgments We thank Dr. Tong Wu for his implementation and testing of the Monte Carlo simulation program. This research is supported in part by Grant #1262933 and #1560037 from the National Science Foundation.

References

Arridge, S. R. (1999). Optical tomography in medical imaging. *Inverse Problems, 15*, R41–R93.

Asrafi, M., Liu, H., & Kuo, F-C. (2011). On testing effectiveness of metamorphic relations: A Case study, *In 5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pp. 147–156.

Baresi L., & Young, M. (2001). Test oracles. Technical Report CIS-TR01 -02, Department of Computer and Information Science, University of Oregon.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The Oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering, 41*(5), 507–525.

Chan, W. K., Cheung, S. C., & Leung, K. R. (2007). A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research, 4*(1), 60–80.

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

Chen, T.Y., Feng, J., Tse, T.H. (2002). Metamorphic testing of programs on partial differential equations: a case study. *In Proceedings of 26th Annual International Computer Software and Applications Conference (COMPSAC)*, pp. 327–333.

Chen, T. Y., Ho, J. W. K., Liu, H., & Xie, X. (2009). An innovattive approach for testing bioinformatics programs using metamorphic testing, BMC Bioinformatics, pp. 10–24.

Chen, T. Y., Huang, D. H., Tse, T. H., & Zhou, Z. Q. (2004). 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*, pp. 569–583.

Chen, C., Lu, J. Q., Li, K., Zhao, S., Brock, R. S., & Hu, X. H. (2007). Numerical study of reflectance imaging using a parallel Monte Carlo method. *Medical Physics, 34*, 2939–2948.

Chen, T. Y., Poon, P. L., & Xie, X. (2015). METRIC: METamorphic relation identification based on the category-choice framework. *Journal of Systems and Software, 116*(C), 177–190.

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 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 191–195.

Confocal microscope, https://en.wikipedia.org/wiki/Confocal_microscopyWikipedia, last accessed on April 30, 2016.

Ding, J., & Zhang D. (2016). A machine learning approach for developing test Oracles for testing scientific software,*In the 28th SEKE (SEKE 2016)*, San Francisco, July 1–3.

Ding, J., Clarke, P. J., Argote-Garcia, G., & He, X. (2009). A methodology for evaluating test coverage criteria of high level Petri nets. *Information and Software Technology, 51*(11), 1520–1533.

Ding, J., Wu, T., Lu, J. Q., Hu, X. (2010). Self-checked metamorphic Testing of an image processing program, *The 4th IEEE International Conference on Security Software Integration and Reliability Improvemen*t, Singapore, June 9–11.

Ding, J., Zhang, D., Hu, X. (2016). An application of metamorphic testing for testing scientific software, *In 1st workshop on metamorphic testing with ICSE*, Austin, TX, May 16.

Farrell, P. E., Pigott, M. D., Gorman, G. J., Ham, D. A., Wilson, C. R., & Bond, T. M. (2011). Automated continuous verification for numerical simulation. *Geoscientific Model Development, 4*(2), 435–449.

Gotleib, A. & Botella, B. (2003). Automated metamorphic testing, In *Proceedings of 27th Annual International Computer Software and Applications Conference*, (pp. 34–40).

Gotlieb, A., Botella, B., & Rueher, M. (1998). Automatic test data generation using constraint solving techniques. *In ACM International Symposium on Software Testing and Analysis (ISSTA). Software Engineering Notes*, 23(2):53–62.

Guderlei, R., & Mayer, J. (2007). Statistical metamorphic testing—testing programs with random output by means of statistical hypothesis tests and metamorphic testing", *In Proceedings of the 7th International Conference on Quality Software*. pp. 404–409, 2007.

Hu, P., Zhang, Z., Chan, W. K., & Tse, T. H. (2006). An empirical comparison between direct and indirect test result checking approaches. *In Proceedings of the 3rd International Workshop on Soft. Quality Assurance*, pp. 6–13.

Jameel, T., Lin, M., & Chao, L. (2015). Test oracles based on metamorphic relations for image processing applications. *In 16th International Conference on SE, AI, Networking and Parallel/Distributed Computing (SNPD),* pp. 1–6.

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *In IEEE Transactions on Software Engineering*, *37*(5), pp. 649–678, September–October 2011.

Kanewala, U., & Bieman, J. M. (2014). Testing scientific software: A systematic literature review. *Information and Software Technology, 56*(10), 1219–1232.

Kanewala, U., Bieman, J. M., & Ben-Hur, A. (2015). Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. *Journal of Software Testing, Verification and Reliability, 26*(3), 245–269.

Keijzer, M., Jacques, S. L., Prahl, S. A., & Welch, A. J. (1989). Light distributions in artery tissure: Monte Carlo simulations for finite-diameter laser beams. *Lasers in Surgery and Medicine, 9*, 148–154.

Le, V., Afshari, M., & Su, Z. (2014). Compiler validation via equivalence modulo inputs. *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, pp.216–226.

Lindvall, M., Ganesan, D., Árdal, R., & Wiegand, R. E. (2015). "Metamorphic model-based testing applied on NASA DAT: an experience report". *In Proceedings of the 37th International Conference on Software Engineering*, *Vol. 2* (ICSE '15), Vol. 2. pp. 129–138.

Liu, H., Kuo, F., Towey, D., & Chen, T. Y. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering, 40*(1), 4–22.

Mayer, J., Guderlei, R. (2006). On random testing of image processing applications. *In Proceedings of 6th International Conference on Quality Software (QSIC'06)*, pp. 85–92.

Mayer, J. & Guderlei, R. (2006). An empirical study on the selection of good metamorphic relations. *In Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC),* pp. 475–484.

Mayer, J., Infor, A. A., Ulm, U. (2005). On testing image procesing applications with staticstical methods. Software Engineering (SE 2005), *Lecture Notes in Informatics*, pp. 69–78.

Murphy, C. Kaiser, G., Hu, L. & 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)*, pp. 867–872.

Murphy, C. (2010). Metamorphic testing techniques to detect defects in applications without test Oracles. Doctoral dissertation, Columbia University.

Murphy, C., Shen, K.,& Kaiser, G. (2009). Automatic system testing of program without test oracles. *In Proceedings of 2009 ACM International Symposium of Software Testing and Analysis (ISSTA).*

Murphy, C., Shen, K., & Kaiser, G. (2009). Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles, *In Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation (ICST).*

Nguyen-Hoan, L., Flint, S., Sankara, R. (2010). A survey of scienfitic software development, *In 2010 ACM International Symposium on Empiricial Software Engineering and Measurement (ESSM'10),* pp. 12:1–12:10.

Nie, C., & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Survey, 43*(2), 11.

Obayashi, M., Kubota, H., McCarron, S. P., & Mallet L. (1998). The assertion based testing tool for OOP: ADL2. http://adl.xopen.org/exgr/icse/icse98.htm May 1998.

Pacheco, P. (1996). Parallel Programming with MPI. Morgan Kaufmann; 1st edition.

Pezzè, M., & Young, M. (2007). *Software testing and analysis: Process, principles, and techniques*. New Jersey: Wiley.

Segura, S., Fraser, G., Sanchez, A., & Ruiz-Cortes, A. (2016). A Survey on Metamorphic Testing. In *IEEE Transaction on Software Engineering* (vol. PP, no. 99). doi:10.1109/TSE.2016.2532875.

Sanders, R., Kelly, D. (2008). The challenge of testing scientific software.*In Proceedings of the Conference for the Association for Software Testing (CAST),* pp. 30–36, Toronto, July.

Shan, L. & Zhu, H. (2009). Generating structually complex test cases by data mutation: A case study of testing an automated modelling tool. The Computer Journal, *52*(5).

Weyuker, E. J. (1982). On testing non-testable program. *Computer Journal, 25*(4), 465–470.

Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., & Chen, T. Y. (2011). Testing and validating machine learning classifiers by metamorphic testing. *J. System and Software., 84*(4), 544–558.

Zhou, Z. Q., Chan, W. K., Chan, W. K., Tse, T. H., & HU, P. (2009). Experimental study to compare the use of metamorphictesting and assertion checking. *Journal of Software, 20*(10), 2637–2654.

Zhou, Z.Q., Xiang, S., Chen, T.Y. (2015). Metamorphic testing for software quality assessment: A study of search engines. *In IEEE Transactions on Software Engineering*, PrePrints, doi:10.1109/TSE.2015.2478001.

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

Zhu, H., & He, X. (2002). A methodology of testing high-level petri nets. *Journal of Information and Software Technology, 44*, 473–489.

Junhua Ding is an associate professor of computer science with East Carolina University (ECU). He received BS, MS, and Ph.D., all in computer science in 1994, 1997, and 2004, respectively. Prior he joined ECU at 2007, he had worked as a software engineer and project manager with medical companies for 8 years. His research interests are software design and analysis, software testing, Petri nets, and cytometry. He has published 60 peer-reviewed conference proceeding and journal papers. He is a member of ACM and IEEE.

Xin-Hua Hu is a professor of physics with East Carolina University (ECU). He received BS and MS in Physics from Nanka University in 1982, 1985, respectively, and Ph.D. in physics from University of California at Irvine in 1991. His research is mainly focused on flow cytometry study of biological cells through light scattering and turbid biological tissues to determine their optical properties (http://bmlaser.physics.ecu.edu). In recent years, he has developed a new method of polarization diffraction imaging flow cytometry that allows rapid measurement of the coherent light distribution from flowing cells and extracts quantitative information of cellular morphology for cell assay and phenotyping. He has published more than 50 papers in peer-reviewed journals, and more than 60 conference proceeding papers and presentations.