Metamorphic Testing: A Simple yet Effective Approach for Testing Scientific Software

Upulee Kanewala
Gianforte School of Computing, Montana State University.

Tsong Yueh Chen
Department of Computer Science and Software Engineering, Swinburne University of Technology.

Testing scientific software is a difficult task due to their inherent complexity and the lack of test oracles. In addition, these software systems are usually developed by *end user developers* who are neither normally trained as professional software developers nor testers. These factors often lead to inadequate testing. *Metamorphic testing* is a simple yet effective testing technique for testing such applications. Even though MT is a well-known technique in the software testing community, it is not very well utilized by the scientific software developers. The objective of this article is to present MT as an effective technique for testing scientific software. To this end, we discuss why MT is an appropriate testing technique for scientists and engineers who are not primarily trained as software developers. Especially, how it can be used to conduct systematic and effective testing on programs that do not have test oracles without requiring additional testing tools.

Index Terms—Metamorphic testing, Scientific software, Quality assurance, Software testing

WHAT MAKES TESTING SCIENTIFIC SOFTWARE DIFFICULT?

Scientific software is widely used for making critical decisions in various scientific and engineering domains. For example, simulations are often used in place of physical experiments due to the time and cost constraints associated with conducting physical experiments. Further, decisions made by these software systems can affect day-to-day human life such as predictions made by climate models. Thus, it is important to make sure that these software systems are producing the correct results. Previous studies have reported many instances where scientific software systems were affected by faults such as seismic programs loosing precision due to one-off errors,[1] compromised performance in coordinate measuring machine (CMM) due to software faults[2] and geoscience software systems producing seemingly correct yet different results that are hard to categorize as incorrect.[3] Previous work also report situations where software faults cause retractions of published work.[4] Testing is the most widely used approach for quality assurance of software. But some inherent characteristics in scientific software make it difficult to conduct systematic testing in these programs: [5]

- **Correct answers are often unknown.** Typically, scientific software is exploratory in nature and due to this the correct results are often unknown. If the result is known there would be no need to develop the software. In such situations only bounds or ranges of solutions might be available. Typically, in testing an expected output is used to decide test case passing or failing. This would make it challenging to conduct systematic testing in these programs.
- **Practically difficult to validate the computed output.** Scientific software often implements mathematical models that involves complex calculations. Further, they tend to produce complex outputs. Both these characteristics make it hard to determine the correctness of the produced output of the software. This makes it challenging to use automated test case generation approaches such as random test generation since the output of such test cases are difficult to validate.
- **Inherent uncertainties.** Often scientific software is written to simulate models with inherent uncertainties. For some of these scientific programs, there may be more than one possible output. This makes it challenging to conduct testing on these programs.
- **Choosing suitable tolerances.** Scientific software systems often involve complex floating-point computations. Thus, specifying the acceptable tolerance for the expected output in test cases is difficult.
- **Incompatible testing tools.** Programing languages such as FORTRAN are widely used in the scientific community for developing scientific software. But usually testing tools are developed for languages such as JAVA and C++ that are commonly used by the software engineering community. Thus, these testing tools are not effective for testing scientific programs.

WHAT IS METAMORPHIC TESTING (MT)?

Consider a program that will accept a list of real numbers and compute their average. Suppose that the input list has 2 million real numbers. How can we know the returned average is correct or not? Though we are not able to validate the computed average in this case, we do know some relationships between the outputs of some related inputs. For example, consider a new list of real numbers, which is a permuted list of the original list of real numbers, or which consists of 4 million real numbers by duplicating the original list of real numbers. For either of the new lists of real numbers, the new average is expected to be the same as the old average (subject to some round off error tolerance). If the new and old averages are not the same, then we know that the program of computing average has bugs. This is the intuition of metamorphic testing.

In software testing, passing or failing of a test case is decided using a test oracle and it is an essential component for conducting systematic testing. Metamorphic testing (MT) uses metamorphic relations (MRs) to determine whether a test case has passed or failed. A MR specifies how the output of the program is expected to change when a specified change is made to the input. The following is the typical process for applying MT to a given program and Figure 1 depicts this process:[6]

1. MR identification: identifying MRs for the program under test can be done based on the specification.
2. Source test case creation and execution: commonly used test generation techniques such as random, structural coverage or fault-based test input generation can be used. Then the generated source test cases are executed on the program under test.
3. Follow-up test case creation: use the MRs identified in Step 1 to transform the source test case to obtain the follow-up test case.
4. Follow-up test case execution: Execute the follow-up test case and compare the outputs of the source and follow-up test cases to verify whether the corresponding MRs are satisfied. Violation of a MR indicates that the program under test is faulty.

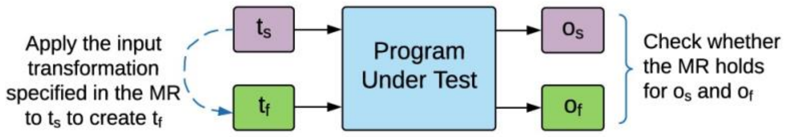
Apply the input transformation specified in the MR to t_s to create t_f
t_s
t_f
Program Under Test
o_s
o_f
Check whether the MR holds for o_s and o_f

Figure 1. Metamorphic testing process. t_s: source test case, t_f : follow-up test case, o_s: output of the source test case and of : o_f: output of the follow-up test case.

Thus, MT checks whether relationships between inputs and outputs of multiple executions were preserved during the program execution and can be used without knowing the correctness of the output for individual executions.

```
@Test
public void testMatrixMultiply() {

        //inputs for the source test case.
        //A and B can be provided by the user or can be randomly generated
        Float64Matrix A = Float64Matrix.valueOf(...);
        Float64Matrix B = Float64Matrix.valueOf(...);

        //Executing the source test case on the matrix multiplication function
        Float64Matrix Os=A.times(B);

        //creating inputs for the follow-up test case
        Random r=new Random();
        Float64Matrix B1 = Float64Matrix.valueOf(new double[][]{
       {r.nextDouble(), r.nextDouble(), r.nextDouble()},
       {r.nextDouble(), r.nextDouble(), r.nextDouble()},
       {r.nextDouble(), r.nextDouble(), r.nextDouble()}});
        Float64Matrix B2=B.minus(B1);

        //Executing the follow-up test cases
        Float64Matrix Of1=A.times(B1)
        Float64Matrix Of2=A.times(B2)

        //Checking whether the metamorphic relation holds
        //between the source and follow-up outputs.
        assertTrue(Os.equals(Of1.plus(Of2)));
}
```

Figure 2. A JUnit test script that uses MT approach to test a matrix multiplication function in the JScience library

Consider a program P that multiplies two matrices A and B. Assume that the result of multiplying A with B is C. Matrix multiplication has the following property: $A \times B = A \times B_1 + A \times B_2$ where $B = B_1 + B_2$. We can use this property as a MR to conduct MT on P. For example, Figure 2 shows a test script written using JUnit to conduct automated testing on the matrix multiplication function in the JScience Matrix class (http://jscience.org/api/org/jscience/mathematics/vector/Matrix.html).

The source test case (which consists of A and B) can be provided by the user or can be generated randomly. For example, assume that the user provided the following simple two matrices:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \; and \; B = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$$

This source test case is executed on the matrix multiplication function under test first. Next, based on the input relationship specified by the above MR, two follow-up test cases are created, namely the test case consisting of A and B_1 and the test case consisting of A and B_2. Here B_1 is randomly generated and B_2 is defined as $B - B_1$. Suppose that

$$B_1 = \begin{bmatrix} 1 & 6 \\ 3 & 5 \end{bmatrix}$$

Then,

$$B_2 = B - B_1 = \begin{bmatrix} 1 & -5 \\ 0 & -1 \end{bmatrix}$$

As shown in Figure 3, these two follow-up test cases are also executed on the matrix multiplication function under test. Finally, the outputs of the source test and the follow-up test cases are validated against the above MR. As shown with this test script, this MR based testing approach allows to generate follow-up test cases automatically and verify relationships between multiple outputs without any manual intervention. Readers who are interested to know more about MT, may consult the article by Chen et. al.[7]

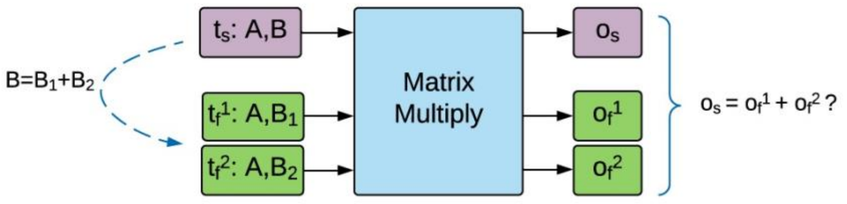

Figure 3. Metamorphic testing of a matrix multiplication program.

WHY USE MT FOR TESTING SCIENTIFIC SOFTWARE?

- Scientific software is often written by scientists who have the domain knowledge required to develop them. However, scientists might lack the knowledge to apply different forms of conventional testing methods. But, as evident from the example in Figure 2, MT is simple in concept, and hence could be easily learned and applied without any prior knowledge of software testing or without any software testing experience.[8]
- As shown in the example given in Figure 2, MT is easy to implement: test scripts could be easily prepared by the scientific software developers to automate the testing process or to incorporate MT into existing testing infrastructures such as JUnit. Thus, MT does not require the developers to buy or maintain additional expensive testing tools.
- As we discussed in Section 2, many scientific and engineering applications face the test oracle problem. This makes it challenging to conduct automated systematic testing on these programs. MT supports automated systematic testing on such programs.

- MT uses MRs to determine whether test cases pass or fail. Often scientific software is developed by domain experts who know the properties of these algorithms the best. Thus, it would be easy for them to derive MRs for testing these programs.
- Scientific software developers will be able to identify the most effective MRs and prioritize them to test their programs using their domain knowledge. For example, consider a program that computes the *sine* value of a given angle x. We can derive the following two MRs for testing the sine function based on its properties: MR1: $\sin(x') = \sin(x)$ where $x' = x + 360°$. MR2: $\sin(x') = -\sin(x)$ where $x' = -x$. Due to the constraints on the testing budget, suppose that we can conduct testing with only one MR. In such a situation, an electrical engineer will most likely choose MR1 to test her program due to the periodicity of current. But, on the other hand a land surveyor may choose MR2 since she usually works with positive and negative angles to represent clockwise and anticlockwise measurements of angles.
- MT provides an effective way to conduct unit testing for scientific software. One of the reasons for the lack of unit testing in scientific software is the difficulty in validating the expected output of the unit for a randomly generated test input. In such situations MT can be used to conduct automated unit testing by means of MRs.
- Scientists often conduct testing using a limited number of test cases with known outputs that they obtain from experiments or analytical solutions. MT can be used to effectively extend these limited number of test cases by deriving MRs and creating follow-up test cases according to the MRs. These follow-up test cases are most likely to execute parts of the program that might not have been executed with the original set of test cases. Thus, MT provides a way to extend existing test cases.
- Many scientific software involves elements of randomness which makes testing difficult. MT can still be applicable in such situations.

SOME EXAMPLES

Testing epidemiological model implementations using MT

Pullum et. al used MT to verify and validate an epidemiological model implementation.[9] Such implementations are used to model how diseases are spread in populations. Thus, it is important to verify and validate these models since they will be used to make critical decisions during a disease spread. Epidemiological model implementations face the oracle problem because these programs are written to find the answer in the first place. Therefore, developing an oracle for testing these programs is practically difficult. One of the approaches used to test these models is to compare the output of the model with data obtained from real phenomena. Obviously, such data is limited. Other approaches used to test this type of programs include comparing the output with results obtained from mathematical models and comparing the results with other simulation models. These techniques are not sufficient for conducting systematic and comprehensive testing on these programs.

The authors tested an ordinary differential equation based epidemiological model and an agent based epidemiological model using MT. They used the data from the 1918 Influenza outbreak to calibrate the models. The authors defined 11 MRs based on making changes to various model parameters and the expected effects that those changes would have on the model output. These MRs were defined using the authors' domain knowledge about these models. Through MT, authors identified an error in the output method of the agent based epidemiological model.

Using MT to conduct automated unit testing on a Small Angle X-ray Scattering (SAXS) program

We used MT to conduct automated unit testing on an open source program written to analyze small angle x-ray scattering data called SAXS.[10] This program reconstructs macromolecular structures using scattering patterns obtained from experiments. This program was initially tested by running the program on a selected set of inputs where the correctness of the produced outputs was determined by domain experts. However, when we showed the domain experts the outputs generated by versions of the program injected with faults, they were unable to identify that the outputs were produced by a faulty version of the program.

Here we report the results of conducting automated unit testing on the following functions that perform several main calculations in the SAXS program:

- *calculateDistance (f1)*: computes distance between atoms
- *findGyrationRadius (f2)*: computes gyration radius of groups of atoms
- *scatterSample (f3)*: main function responsible for scattering

We used the machine learning based MR prediction approach proposed by Kanewala et. al[11] to predict the likely MRs for these functions. The test inputs were generated randomly. There were no violations of these predicted MRs when applied to these three functions.

To evaluate the effectiveness of MT for conducting unit testing, we created faulty versions, known as *mutants*, of these functions using the μJava (https://cs.gmu.edu/_offutt/mujava/) mutation engine. This mutation engine creates mutants of the program by making a syntactic change in the source code. With MT, we say that a mutant is killed if a MR is violated when the corresponding source and follow-up test cases are executed on that mutant. Therefore, the fault detection effectiveness of MT can be measured by the number of mutants killed during the MT process. Obviously, the higher the percentage of mutants killed, the more effective MT is in revealing bugs of a program. We use this process to evaluate the fault detection effectiveness of MT in the functions mentioned above.

Table 1 shows the percentage of mutants killed through MT for individual functions. Overall, 90% of the mutants could be killed using MT. The important thing to note here is that the entire unit testing process was fully automated starting with MR identification, source test case generation, test execution and further, did not require the domain experts to evaluate the correctness of the test outputs. Though no violations of MRs were detected for SAXS, MT helps to establish our confidence on the quality of the SAXS program.

TABLE 1. Mutants detected by predicted MRs. f_1: calculateDistance, f_2: findGyrationRadius, and f_3: scatterSample

	f_1	f_2	f_3	**Total**
No. of faulty versions used	19	54	139	212
No. of faulty versions detected by MT	19	45	127	191
% of detected faulty versions	100	83	91	90

Testing a Monte Carlo simulation program with MT

Ding and Hu[12] used MT for testing a Monte Carlo modeling program that simulates photon propagations in biological tissues for the purpose of accurate generation of reflectance images. The biggest challenge for testing this program is the lack of test oracles. One solution is to compare the results of the Monte Carlo simulation program to experimental results. But, as with many scientific software, building the necessary infrastructure to conduct the relevant physical experiments is time consuming and expensive. For example, in this specific case, conducting a physical experiment would require a laser beam that would produce a specific number of photons, an environment without interruptions from other light sources and good reactive imaging cameras. Thus, the authors used MT to conduct testing on this program.

The authors identified five MRs for the program based on domain knowledge and experimental results. They generated tests that cover all the branches and functions in the program. Through the violation of one of the MRs used for testing, the authors discovered faults in the program and corrected it.

They further evaluated the effectiveness of MT using mutants. The authors created 150 mutants for the Monte Carlo simulation program and they were able to detect 90% (135) of these mutants using MT.

SUMMARY

Some characteristics in scientific software such as, not knowing the correct answers and inherent uncertainties in calculations, make testing them difficult. MT can be an effective testing technique to test these programs. Instead of checking the correctness of individual test outputs, MT checks whether the changes in the test outputs are according to what is expected by the program with respect to the changes in the inputs. These relationships between inputs and the expected changes in the outputs are referred as MRs. Scientists, who typically develop these scientific software, would be in a great position to identify effective MRs because of their domain knowledge and, thus would be able to effectively test their software using MT. MT has been successfully applied for testing various scientific software including epidemiological model implementations, small angle x-ray scattering programs and Monte Carlo simulations. We are strongly confident that MT is one of the most appropriate and cost-effective testing techniques for scientists and engineers.

REFERENCES

1. L. Hatton, “The T experiments: errors in scientific software,” IEEE Computational Science Engineering, vol. 4, no. 2, pp. 27 –38, Apr.– Jun. 1997.
2. A. J. Abackerli, P. H. Pereira, and N. Calônego Jr., “A case study on testing CMM uncertainty simulation software (VCMM),” Journal of the Brazilian Society of Mechanical Sciences and Engineering, vol. 32, pp. 8 – 14, Mar. 2010.
3. L. Hatton and A. Roberts, “How accurate is scientific software?” IEEE Transactions on Software Engineering, vol. 20, no. 10, pp. 785 –797, Oct. 1994.
4. G. Miller, “A scientist’s nightmare: Software problem leads to five retractions,” Science, vol. 314, no. 5807, pp. 1856–1857, 2006. [Online]. Available: http://www. sciencemag.org/content/314/5807/1856.short
5. U. Kanewala and J. M. Bieman, “Testing scientific software: A systematic literature review,” Information and Software Technology, vol. 56, no. 10, pp. 1219 – 1232, 2014.
6. S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés, “A survey on metamorphic testing,” IEEE Trans. Software Eng., vol. 42, no. 9, pp. 805–824, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2532875
7. T. Y. Chen, F.-C. Kuo, H. Liu, P. L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” ACM Computing Surveys (in press), 2017.
8. T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, “An effective testing method for end-user programmers,” in Proceedings of the First Workshop on End-user Software Engineering (WEUSE 2005), 2005, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083236
9. L. L. Pullum and O. Ozmen, “Early results from metamorphic testing of epidemiological models,” in 2012 ASE/IEEE International Conference on BioMedical Computing (BioMed-Com), Dec 2012, pp. 62–67.
10. http://cgi.cs.arizona.edu/~mstrout/Projects/SAXS/software.php, 2011 (accessed September 11, 2017).
11. U. Kanewala, J. M. Bieman, and A. Ben-Hur, “Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels,” Software Testing, Verification and Reliability, vol. 26, no. 3, pp. 245–269, 2016, stvr.1594. [Online]. Available: http://dx.doi.org/10.1002/stvr.1594

12. J. Ding and X. Hu, “Application of metamorphic testing monitored by test adequacy in a monte carlo simulation program,” Software Quality Journal, vol. 25, no. 3, pp. 841–869, 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9337-3

ABOUT THE AUTHORS

Upulee Kanewala is an Assistant Professor at Montana State University, USA. Her research interests include software testing, metamorphic testing and quality assurance of scientific software. Upulee received her Ph.D. in Computer Science from Colorado State University in 2015, a Master of Science in Computer Engineering from Purdue University in 2010, and a Bachelor of Science in Computer Engineering from University of Peradeniya, Sri Lanka in 2007. She has published multiple peer-reviewed articles and book chapters on Metamorphic Testing including the first paper on automatic detection of Metamorphic Relations. Her address is Gianforte School of Computing, 357 Barnard Hall, Montana State University, Bozeman, MT 59717. Her email is upulee.kanewala@montana.edu

Tsong Yueh Chen is a Professor at Swinburne University of Technology, Australia. His main research interest is software testing. He got his BSc and MPhil from The University of Hong Kong, MSc from University of London, DIC from the Imperial College, and PhD from The University of Melbourne. Prior to joining Swinburne, he taught at The University of Hong Kong and The University of Melbourne. He is the inventor of Metamorphic Testing and Adaptive Random Testing. His contact address is: Department of Computer Science and Software Engineering, Swinburne University of Technology, Vic 3122, Australia. Email is tychen@swin.edu.au