This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

Metamorphic Testing: Testing the Untestable

Sergio Segura
Department of Computer Languages and Systems, University of Seville, Spain.

Dave Towey
School of Computer Science, University of Nottingham Ningbo China, China.

Zhi Quan Zhou
Institute of Cybersecurity and Cryptology, School of Computing and Information Technology, University of Wollongong, Australia.

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

ABSTRACT. What if we could know that a program is buggy, even if we could not tell whether or not its observed output is correct? This is one of the key strengths of metamorphic testing, a technique where failures are not revealed by checking an individual concrete output, but by checking the relations among the inputs and outputs of multiple executions of the program under test. Two decades after its introduction, metamorphic testing has become a fully-fledged testing technique with successful applications in multiple domains, including online search engines, autonomous machinery, compilers, Web APIs, and deep learning programs, among others. This article serves as a hands-on entry point for newcomers to metamorphic testing, describing examples, possible applications, and current limitations, providing readers with the basics for the application of the technique in their own projects.

Keywords: *Software testing, metamorphic testing, oracle problem, test case generation.*

Introduction

Suppose you are helping your son with his homework. You ask him how many exercises he has to do in total, and he answers two. You are not sure if that is correct, so a few minutes later you ask how many *math* exercises the teacher has asked him to do, and this time he answers four. This is wrong—the total number of exercises cannot be less than the number of math exercises—and so the boy seems forgetful. Note that there was no need to know if either of the answers were correct to detect the problem and, more importantly, the boy has exposed himself!

Like the child in the example above, some programs are extremely difficult to test because of the lack of an *oracle*—a mechanism that can decide whether or not the program's output is correct in a reasonable amount of time [1]. Consider, for example, testing programs such as compilers, search engines, machine learning systems, or simulators: determining the correctness of the output for a given input may be non-trivial and error-prone. This is known as the *oracle problem*, and the programs suffering from it are often referred to as *non-testable* (or *untestable*) [1]–[3].

Metamorphic testing (MT) is an effective technique for alleviating the oracle problem, and thus for testing "untestable" programs where, as in the previous example, failures are

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

not revealed by checking individual outputs, but by checking expected relations among multiple executions of the program under test. Since its introduction by Chen et al. in 1998 [4], the literature on MT has grown impressively, and many successful applications of the technique have come to light. Some of these successes include the detection of bugs in real-world systems such as the search engines Google and Bing [5], the compilers GCC and LLVM [6], commercial code obfuscators [7], NASA systems [8], and the Web APIs of Spotify and YouTube [9]. Recently, GraphicsFuzz, a spinout company from Imperial College London, has commercialized this technique [10]. In this article, we present an intuitive introduction to metamorphic testing, giving practical examples, describing success stories, and discussing its limitations.

Metamorphic testing in a nutshell

MT approaches the software testing problem from a perspective not used by most other testing strategies: rather than focusing on each individual output, MT looks at *multiple* executions of the program. It checks whether the inputs and outputs of these multiple executions satisfy certain *metamorphic relations*, which are necessary properties of the intended program's functionality. A metamorphic relation transforms existing (source) test cases into new (follow-up) ones. If the program's behavior across these source and follow-up test cases violates the metamorphic relation, the program must be faulty.

As an example, consider the program $\mathrm{merge}(L_1, L_2)$ that merges two lists into a single ordered list without duplicated elements. Deciding whether the output of the program is correct for any two non-trivial input lists is difficult, and thus this is an instance of the oracle problem. However, the order of the parameters should not influence the result, which can be expressed as the following metamorphic relation: $\mathrm{merge}(L_1, L_2) = \mathrm{merge}(L_2, L_1)$. This metamorphic relation can be instantiated into one or more metamorphic tests by using specific input values and checking whether the relation holds, e.g. $\mathrm{merge}([a,d],[k,t]) = \mathrm{merge}([k,t],[a,d])$. If the relation is violated, we could be certain that the program is faulty. In this example, $([a,d],[k,t])$ is called the *source test case*, and $([k,t],[a,d])$ is called the *follow-up test case*. Note that many others metamorphic relations could be identified, for example, $\mathrm{merge}(L_1, L_2) = \mathrm{merge}(L_3, L_4)$, where $L_3 = L_1 + L_1$ and $L_4 = L_2 + L_2$, and $+$ is the list concatenation operator.

MT is also regarded as an effective test data generation technique. This is because a metamorphic relation implicitly defines how a given source test case can be transformed into one or more follow-up test cases such that the relation can be checked. In the previous example, for instance, MT could be used together with a random list generator to automatically construct source test cases $(e.g. ([s,t],[k,l,p]))$ and their respective follow-up test cases $([k,l,p],[s,t])$ by swapping the order of the lists. This could continue until a pair that reveals a bug is found $(\mathrm{merge}([s,t],[k,l,p]) \neq \mathrm{merge}([k,l,p],[s,t]))$, or until a maximum timeout is reached.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

MT was originally proposed two decades ago as a technique to reuse successful test cases (those that pass and thus reveal no failures). Since then, MT has thrived, becoming a well-established testing technique with numerous applications in both academia and industry. For a thorough introduction to the technique we refer the reader to two recent surveys on the topic [2], [3], and a recent webinar [11].

A hands-on example

Suppose that you are part of the testing team for the popular website *Booking.com*, which allows users to find potential lodgings according to their preferences. You run an exploratory test by performing a search for accommodation in Rome, which returns 7,378 result items. Is this output correct? Is there anything missing in the result set? Is there any result not meeting the search criteria included in the list? Answering these questions would be extremely time-consuming. This is a clear case of the oracle problem. To alleviate this problem, and to automate the generation of test cases, MT could be employed by applying the following basic steps.

Step 1. Identification of metamorphic relations

Metamorphic relations are generally identified based on our knowledge of the problem domain, the program specification, and/or the user manual. To identify a metamorphic relation, we may think about how certain changes in the program's inputs may be expected to produce certain changes in the program's outputs [12]. For example, Figure 1 shows the interface of *Booking.com* displaying the results of the search for accommodation in Rome. A closer look at this may lead us to several metamorphic relations, for example:

MR$_1$: Perform a search. Then repeat the search adding a budget filter, "*US$100-US$150 per night",* for instance. The result set of the second search (follow-up test case) should be a subset of the result set of the first search (source test case), where no filter was applied.

MR$_2$: Perform a search for hotels with a "*1 star*" rating filter (source test case). Then, repeat the search four times changing the star rating filter to "*2 stars*", "*3 stars*", "*4 stars*", and "*5 stars*", respectively (follow-up test cases). The results sets of the five searches should not contain any common result item, since the same hotel cannot have two different star ratings at the same time.

MR$_3$: Perform a search (source test case). Then, repeat the search changing the ordering criterion from default ("*Our top picks*") to "*Review score*" (follow-up test case). Both searches should return the same items, regardless of their ordering.

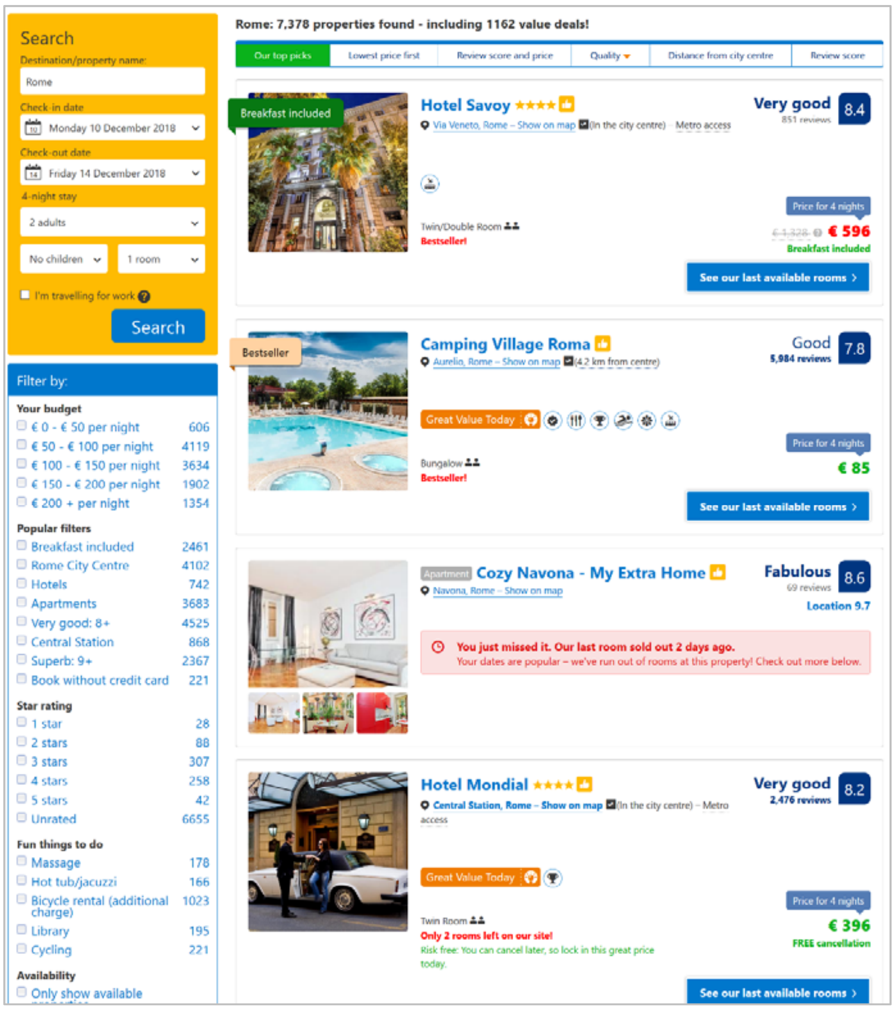
These are just a few of the potentially huge number of metamorphic relations that could be identified for *Booking.com* and similar websites and APIs [5], [9]. The reader may like to try identifying some other relations by looking at the "Popular" filters shown on the left side of Figure 1.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Step 2. Implementation

Once the relations are identified, it is time to implement and run the actual metamorphic tests by instantiating each relation with specific test data. This can be done using any standard unit testing framework. As an example, Figure 2 shows a metamorphic test for the relation MR_1 written in the JUnit framework. Note that the key difference compared with standard unit test cases is that the method under test (`Booking.search`) is executed twice (lines 17 and 24), instead of just once, and that the assertion (line 27) refers to the output of both calls, instead of to a single output.

Several points are worth emphasizing in this example. First, recall that every metamorphic test starts from an existing test case, called the source test case. Source test cases can be designed from scratch, using any standard test design technique, or they can be reused from an existing test suite. Second, note that each metamorphic relation can be typically instantiated into many metamorphic tests, by using different test data. Thus, we could easily implement many other metamorphic tests from MR_1 by simply using different search queries and budget filters.



Rome
Monday 10 December 2018
Friday 14 December 2018
4-night stay
2 adults
No children
1 room
I'm travelling for work

Your budget	
€ 0 - € 50 per night	606
€ 50 - € 100 per night	4119
€ 100 - € 150 per night	3634
€ 150 - € 200 per night	1902
€ 200 + per night	1354
Popular filters	
Breakfast included	2461
Rome City Centre	4102
Hotels	742
Apartments	3683
Very good: 8+	4525
Central Station	868
Superb: 9+	2367
Book without credit card	221
Star rating	
1 star	28
2 stars	88
3 stars	307
4 stars	258
5 stars	42
Unrated	6655
Fun things to do	
Massage	178
Hot tub/jacuzzi	166
Bicycle rental (additional charge)	1023
Library	195
Cycling	221
Availability	
Only show available	

Rome: 7,378 properties found - including 1162 value deals!

Breakfast included
Hotel Savoy ★★★★
Via Veneto, Rome – Show on map (In the city centre) – Metro access
Very good 8.4
851 reviews
Twin/Double Room
Bestseller!
~~€ 1,328~~ € 596
Breakfast included

Bestseller
Camping Village Roma
Aurelio, Rome – Show on map (4.2 km from centre)
Good 7.8
5,984 reviews
Great Value Today
Bungalow
Bestseller!
€ 85

Apartment Cozy Navona - My Extra Home
Navona, Rome – Show on map
Fabulous 8.6
69 reviews
Location 9.7
You just missed it. Our last room sold out 2 days ago.
Your dates are popular – we've run out of rooms at this property! Check out more below.

Hotel Mondial ★★★★
Central Station, Rome – Show on map (In the city centre) – Metro access
Very good 8.2
2,476 reviews
Great Value Today
Twin Room
Only 2 rooms left on our site!
Risk free: You can cancel later, so lock in this great price today.
€ 396
FREE cancellation

Figure 1. Booking.com search interface.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

Step 3. Automated test case generation

Finally, the real potential of MT is fully realized when combining it with automated test data generation techniques. In the example of Figure 2, for instance, both the query search and the filter data could be randomly generated, enabling the construction of a potentially limitless number of test cases. Note that this process would include not only the generation of inputs, but also the generation of the corresponding output assertions, truly achieving full test automation.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class BookingSearchTest {

    @Test
    public void searchMetamorphicTest() {

        // Create query
        BookingQueryObject query = new BookingQueryObject();
        query.setDestination("Rome");
        query.setCheckIn("11/12/2018");
        query.setCheckOut("11/15/2018");
        query.setnAdults(2);

        // Source test case
        BookingSearchResult st = Booking.search(query);

        // Follow-up test case
        BudgetFilter filter = new BudgetFilter();
        filter.setMinBudget(120);
        filter.setMaxBUdget(180);
        filter.setCurrency("US");
        BookingSearchResult fut = Booking.search(query,filter);

        // Metamorphic relation assertion
        assertTrue("Not a subset", fut.isSubset(st));
    }
}
```

Figure 2. Sample implementation of a metamorphic test in JUnit 4.

Approaches for the identification of metamorphic relations

The effectiveness of MT is strongly influenced by the metamorphic relations used, and thus identifying effective metamorphic relations is a critical step. The identification of metamorphic relations is a task that requires creativity and domain knowledge, but there are clues that can help in the process [12]. We next describe two common approaches for the identification of metamorphic relations.

Input-driven approach. This strategy involves thinking of changes to the program's inputs that should produce expected changes in the outputs [12]. The possible changes in the input parameters depend on their data type. For example, possible operations in an input list might include: adding an element to the list; removing an element from the list; splitting the list; reordering the list; and so on. Analogously, possible changes to a search

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

query might involve adding or removing filtering, sorting or pagination-related parameters [5], [9]. These possible changes provide clues about how source test cases could be changed to generate new follow-up test cases, and consequently for the identification of metamorphic relations. This approach is frequently used in numerical and graph-theory programs.

Output-driven approach. In contrast to the previous approach, this method proposes starting from possible relations among outputs typically found in the target domain, and then thinking about what kind of changes in the program's inputs would lead to satisfaction of the expected relation among outputs. For example, typical relations between outputs in search operations are: having a result set which is a subset of another result set; having two result sets containing the same items; or having two disjoint result sets (sets with no common elements) [5], [9]. Suppose we are looking at the *subset* relation between outputs in the *Booking.com* example. We should think of changes to the search query that filters some items out of the result set. For example, we could perform a search for hotels, and then perform the same search for hotels, but this time with *"Pets allowed"* (or any other filter). The result set of the latter search should be a subset of the former (where no filters were used). This approach is frequently used in programs accessing data repositories such as information systems, search engines, and Web APIs.

Regardless of the approach followed, previous studies suggest that metamorphic relations should be diverse, meaning that they should involve different input parameters and input constraints to exercise the program under test as thoroughly as possible [13].

Applications

Figure 3 summarizes the domains where MT has been applied, based on a survey of 84 case studies published between January 1998 and May 2018 (note that the total number of publications on MT, considering all types of papers, is much higher). The most popular domain is web services and applications (14%), followed by computer graphics (11%). We also found a variety of applications to other fields (24%) such as financial software, optimization programs, cybersecurity, and data analytics, as well as industrial applications in organizations such as NASA and Adobe. Interestingly, only 5% of the papers reported results in numerical programs, even though this is a frequently used domain to illustrate how MT works in the literature. The arrows in Figure 3 represent whether there is an increasing or a decreasing trend in the applications in a particular domain in the last three years. The increasing number of applications of MT to bioinformatics and artificial intelligence, including machine learning and autonomous vehicles, is worth noting.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

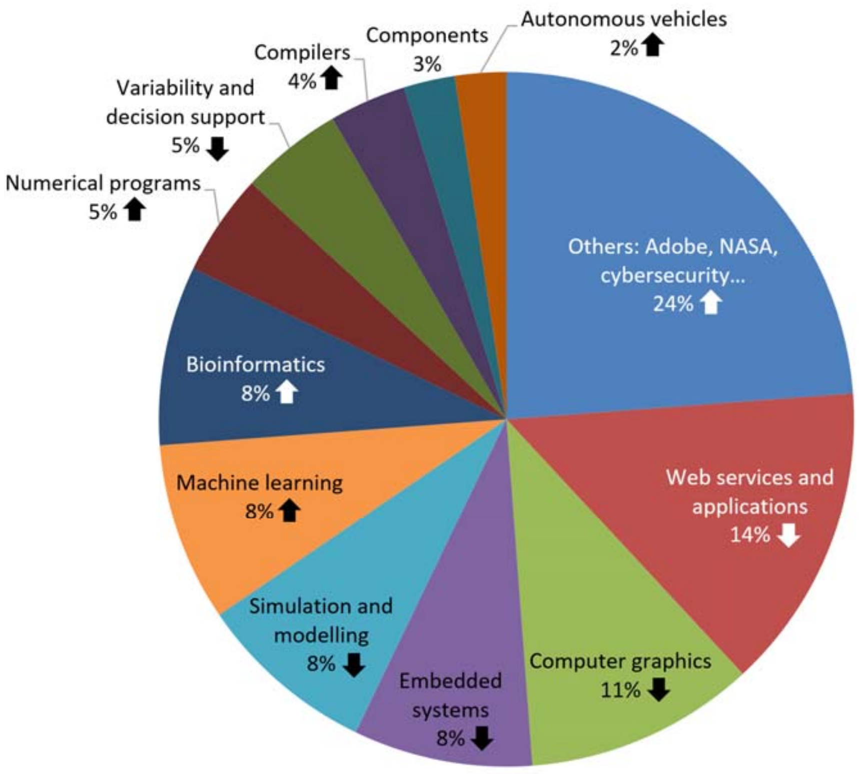


Figure 3. Application domains

We next highlight some successful applications of MT in the domains of compilers and artificial intelligence.

Compilers

Several researchers have proposed using MT to find failures in compilers based on the following idea: removing dead source code from a program should not alter the functionality of the compiler-generated code [6]. The approach works in three steps: 1) Run a program (source test case) with some inputs using a profiler to identify the executed code; 2) Create a new version of the program (follow-up test case) by removing some of the dead code (statements not executed in step 1); and 3) Run the new program on the same inputs, reporting any observed changes in the outputs as a failure. This approach was reported to have detected 147 bugs (110 fixed) in the GNU Compiler Collection (GCC) and Low Level Virtual Machine (LLVM) C compilers when first published [6], but hundreds more bugs have been found (and fixed) since then [14].

A similar strategy has been used in industry by *GraphicsFuzz* for testing graphics drivers [10]. Their approach works as follows [15]: First, an image is rendered using a *shader program* (through the graphics driver under test, which includes a shader compiler that translates the shader program into low-level machine code for the target GPU). The output image is called the original image. Second, a transformation is applied to the shader program that should have no significant impact on how the image is rendered (for

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

example, adding "+0.0" to an arithmetic expression). Third, an image is rendered using the modified shader program (still through the graphics driver under test), obtaining a so-called variant image. Finally, the original and variant images are compared. If the differences are significant it means that the graphics driver is faulty. At the time of writing, the GraphicsFuzz toolset had revealed more than 83 issues in several graphics compilers from popular GPU designers. Among others, it has been publicly credited for detecting bugs in Apple (iOS-Webkit), NVIDIA, and Chrome (on Samsung S6), receiving a Google Chrome bug bounty of $2,000. GraphicsFuzz was acquired by Google in Aug 2018.

Artificial intelligence

MT has been used for testing artificial intelligence (AI) tools such as supervised and unsupervised machine learning programs [16]. These are programs that "learn" from a set of data samples composed of attributes and labels. Metamorphic relations in this domain define changes in the samples used for learning that produce a predictable effect in the knowledge extracted from them: changing the order of the attributes in the samples should have no impact in the outcome, for example. Among others, MT has been used to detect real bugs in the machine learning tools Weka and RapidMiner.

MT has also proved to be effective to test AI-driven systems. Researchers at the Fraunhofer Center for Experimental Engineering (USA) developed a framework for the automated testing of simulated autonomous drones [17]. Their approach starts by defining a model of a flying scenario and observing the drone behavior in this scenario. Then, they programmatically generate multiple variations of the scenario where the outcome of the drone flight should be equivalent. For example, the drone should behave consistently no matter whether it is flying north or south, if the distances and relative position of obstacles are the same. This approach enabled them to detect a number of issues that led to unexpected behavior of the drone, including fatal crashes. For example, they found that the drone had problems landing in some situations when rotating objects in the scene. They determined that the problem was related to the direction of sunlight not being rotated, and that this caused a shadow to fall on the landing-pad in some orientations, causing the vision system to fail to recognize the landing spot.

A similar idea was used by researchers at University of Virginia and Columbia University to implement *DeepTest*, a testing tool for self-driving cars driven by Deep Neural Networks (DNNs) [18]. DeepTest automatically generates synthetic test cases with different driving conditions such as rain or fog, where the car should behave similarly. Among other results, DeepTest found thousands of erroneous actions, under different realistic driving conditions, some of which led to fatal crashes in three top performing DNNs in the Udacity self-driving car challenge. Figure 4 shows one of the test cases generated by DeepTest that revealed a failure [19]. The blue arrow in the left-hand image shows the original trajectory of the car. The red arrow in the right-hand image shows the erroneous trajectory calculated by the DNN-driven system when fog was added to the scene.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

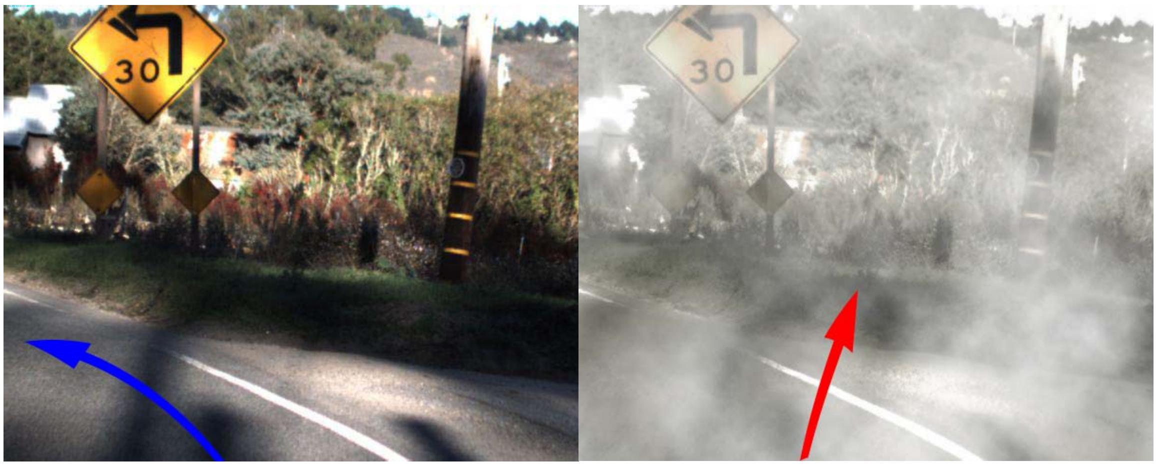


Figure 4. A sample of erroneous behaviour found by DeepTest [19].

Limitations

MT also has some limitations that may narrow its applicability in certain domains. First, it is worth noting that while MT can be used to alleviate the oracle problem, it cannot solve it completely. This is because metamorphic relations cannot be used to tell whether or not the output of a program is the expected one. For instance, the metamorphic test of Figure 2 could pass even if the outputs of the source test case ("hotels in Rome") and the follow-up test case ("hotels in Rome with a budget of $120-$180 per night") are wrong (if both searches return a particular hotel in Florence, for instance). Thus, MT (on its own) may not be suitable for testing critical systems that require the correctness checking of each individual output.

Another limitation of MT is the need to identify metamorphic relations, which is typically a manual process requiring effort and creativity. Although some approaches for the automated discovery of metamorphic relations exist, so far, they have mostly focused on numerical programs [20], [21]. Reported experiences of teaching MT, however, suggest that students can easily identify effective metamorphic relations after only a few hours of training [13].

Finally, the number of metamorphic relations in most non-trivial programs is potentially huge. This leads to a common problem when applying MT: how to select the most effective metamorphic relations. Although some heuristics for guiding the process do exist, as previously explained, more systematic approaches for such optimal selection are yet to be proposed.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

Conclusion

MT is a thriving testing technique with demonstrated ability to address the oracle problem and to enable test case generation. Future contributions are expected in many areas, including new application domains, automated inference of metamorphic relations, tools, and integration with other testing techniques.

References

[1] E.J. Weyuker, “On Testing Non-Testable Programs”, The Computer Journal, vol. 25, no. 4, 465–470, 1982.

[2] S. Segura, G. Fraser, A. Sanchez and A. Ruiz-Cortés, “A Survey on Metamorphic Testing”, *IEEE Transactions on Software Engineering*, vol. 42, 805-824, 2016.

[3] 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*, vol. 51, no. 1, 4:1-4:27, 2018.

[4] T.Y. Chen, S.C. Cheung and S.M. Yiu, “Metamorphic Testing: A New Approach for Generating Next Test Cases”, *Technical Report HKUSTCS98-01*. Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[5] Z.Q. Zhou, S. Xiang and T.Y. Chen, “Metamorphic Testing for Software Quality Assessment: A Study of Search Engines”, *IEEE Transactions on Software Engineering*, vol. 42, no. 3, 264-284, 2016.

[6] V. Le, M. Afshari and Z. Su, “Compiler Validation via Equivalence Modulo Inputs”, in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 216-226, 2014.

[7] T.Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas and Z.Q. Zhou, “Metamorphic Testing for Cybersecurity”, *Computer*, vol. 49, no. 6, 48-55, 2016.

[8] M. Lindvall, D. Ganesan, R. Ardal and R.E. Wiegand, “Metamorphic Model-Based Testing Applied on NASA DAT - An Experience Report”, in *Proceedings of the IEEE/ACM 37th International Conference on Software Engineering*, 129-138, 2015.

[9] S. Segura, J.A. Parejo, J. Troya and A. Ruiz-Cortés, “Metamorphic Testing of RESTful Web APIs”, *IEEE Transactions on Software Engineering*. In press. https://doi.org/10.1109/TSE.2017.2764464

[10] GraphicsFuzz. https://www.graphicsfuzz.com , accessed June 2018.

[11] S. Segura and Z.Q. Zhou, “Metamorphic Testing: Introduction and Applications”, *ACM SIGSOFT Webinar*, 2017. https://event.on24.com/wcc/r/1451736/8B5B5925E82FC9807CF83C84834A6F3D

[12] T.Y. Chen, P. Poon and X. Xie, “METRIC: METamorphic Relation Identification based on the Category-Choice Framework”, *Journal of Systems and Software*, vol. 116, 177-190, 2016.

[13] H. Liu, F.-C. Kuo, D. Towey and T.Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?", *IEEE Transactions on Software Engineering*, vol. 40, no. 1, 4-22, 2014.

[14] Z. Su, Personal website, http://web.cs.ucdavis.edu/~su , accessed June 2018.

[15] A.F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, “Automated testing of graphics shader compilers”, *Proceedings of the ACM on Programming Languages*, vol. 1, issue OOPSLA, 93:1-93:29, October 2017.

[16] X. Xie, J.W.K. Ho, C. Murphy, G. Kaiser, B. Xu and T.Y. Chen, “Testing and Validating Machine Learning Classifiers by Metamorphic Testing”, *Journal of Systems and Software*, vol. 84, 544-558, 2011.

[17] M. Lindvall, A. Porter, G. Magnusson and C. Schulze, “Metamorphic Model-Based Testing of Autonomous Systems”, in *Proceedings of the IEEE/ACM 2nd International Workshop on Metamorphic Testing*, in conjunction with the 39th International Conference on Software Engineering (ICSE '17). 2017.

[18] Y. Tian, K. Pei, S. Jana and B. Ray, “DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars”, in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[19] Y. Tian, K. Pei, S. Jana and B. Ray, DeepTest website on GitHub, https://deeplearningtest.github.io/deepTest , accessed March 2018.

[20] 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, 245-269, 2016.

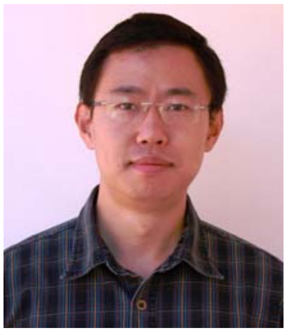
[21] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations", in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 701-712, 2014.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

Sergio Segura is a senior lecturer in software engineering at the University of Seville, Spain. His main research interest is in software testing and search-based software engineering. Segura received a PhD in computer science from the University of Seville. Contact him at sergiosegura@us.es .

Dave Towey is an associate professor in the School of Computer Science at the University of Nottingham Ningbo China. His research interests include software testing, computer security, and technology-enhanced education. Towey received a PhD in computer science from The University of Hong Kong. He is a member of IEEE. Towey is the corresponding author for this article. Contact him at dave.towey@nottingham.edu.cn .



Zhi Quan Zhou is an associate professor in software engineering at the University of Wollongong, Australia. His research interests include software testing and debugging, security testing, machine learning and self-driving cars, and citation analysis. Zhou is one of the few earliest pioneers who opened up and established the research field of metamorphic testing. He received a PhD in software engineering from The University of Hong Kong. Zhou acknowledges the support from the Australian Research Council (project ID: LP160101691) and Suzhou Insight Cloud Information Technology Co., Ltd.. Contact him at zhiquan@uow.edu.au .

Tsong Yueh Chen is a professor of software engineering at Swinburne University of Technology. His main research interest is in software testing. Chen is the inventor of metamorphic testing and adaptive random testing. He received a PhD in computer science from the University of Melbourne. Chen is a Senior Member of IEEE. Contact him at tychen@swin.edu.au .

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2018.2875968, IEEE Software

Tweets:

* Metamorphic testing can detect bugs even when we might not be able to tell if the program output is correct.

* Metamorphic testing can test untestable programs because it uses the program to test itself.

* Metamorphic testing is a fully automatable test case generation and result verification technique.

* Metamorphic testing can test self-driving cars, compilers, cybersecurity, big data and AI software, and much more.

* Google buys GraphicsFuzz, a startup from Imperial College London, to apply metamorphic testing to graphics drivers.

0740-7459 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.