

Metamorphic Testing: Testing the Untestable

Sergio Segura, University of Seville

Dave Towey, University of Nottingham Ningbo China

Zhi Quan Zhou, University of Wollongong

Tsong Yueh Chen, Swinburne University of Technology

// What if we could know that a program is buggy, even if we could not tell whether or not its observed output is correct? Metamorphic testing provides this ability. This article explains the basics of the technique. //



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 fewer than the number of math exercises—so the boy seems forgetful. There was no need to know if the answers

were correct to detect the problem, and, more importantly, the boy revealed the problem himself!

Like the child in this example, some programs are extremely difficult to test because of the lack of an oracle, which is 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 nontrivial and error prone. This is known as the *oracle problem*,

and the programs suffering from it are often referred to as *nontestable* (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 not revealed by checking individual outputs but by checking the 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 GNU Compiler Collection (GCC) and Low-Level Virtual Machine (LLVM) compilers,⁶ commercial code obfuscators,⁷ NASA systems,⁸ and the web application programming interfaces (APIs) of Spotify and YouTube.⁹ Recently, GraphicsFuzz, a spin-off company from Imperial College London acquired by Google, 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.

MT 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 $\text{merge}(L_1, L_2)$ that merges two lists into a single ordered list without duplicated elements. Deciding whether the output of the program is correct for any two nontrivial 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: $\text{merge}(L_1, L_2) = \text{merge}(L_2, L_1)$. This metamorphic relation can be instantiated into one or more metamorphic tests by using specific input values and checking whether the relation holds, for example, $\text{merge}([a, d], [k, j]) = \text{merge}([k, j], [a, d])$. If the relation is violated, we could be certain that the program is faulty. In this example, $([a, d], [k, j])$ is called the *source test case*, and $([k, j], [a, d])$ is the *follow-up test case*. Many other metamorphic relations could be identified, for example, $\text{merge}(L_1, L_2) = \text{merge}(L_3, L_4)$, where $L_3 = L_1 + L_1$, $L_4 = L_2 + L_2$, 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—for example, $([s, j], [k, l, p])$ —and their respective follow-up test cases $([k, l, p], [s, j])$ by swapping the order of the lists. This could continue until a pair that reveals a bug is found, such as $\text{merge}([s, j], [k, l, p]) \neq \text{merge}([k, l, p], [s, j])$, or until a maximum timeout is reached.

MT was originally proposed two decades ago as a technique for reusing successful test cases (those that pass and, thus, reveal no failures). Since then, it 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 accommodations 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:

- MR_1 : Perform a search. Then, repeat the search adding a budget filter, such as “US\$100–US\$150 per night.” 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), for which no filter was applied.
- MR_2 : Perform a search for hotels with the rating filter “1 star” (source test case). Then, repeat the search four times, changing the rating filter to “2 stars,” “3 stars,” “4 stars,” and “5 stars” (follow-up test cases). The result sets of the five searches should not contain any common result item, because 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.

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 with any standard unit-testing framework. As an example, Figure 2 shows a metamorphic test for the relation MR_1 written in the JUnit framework. The key

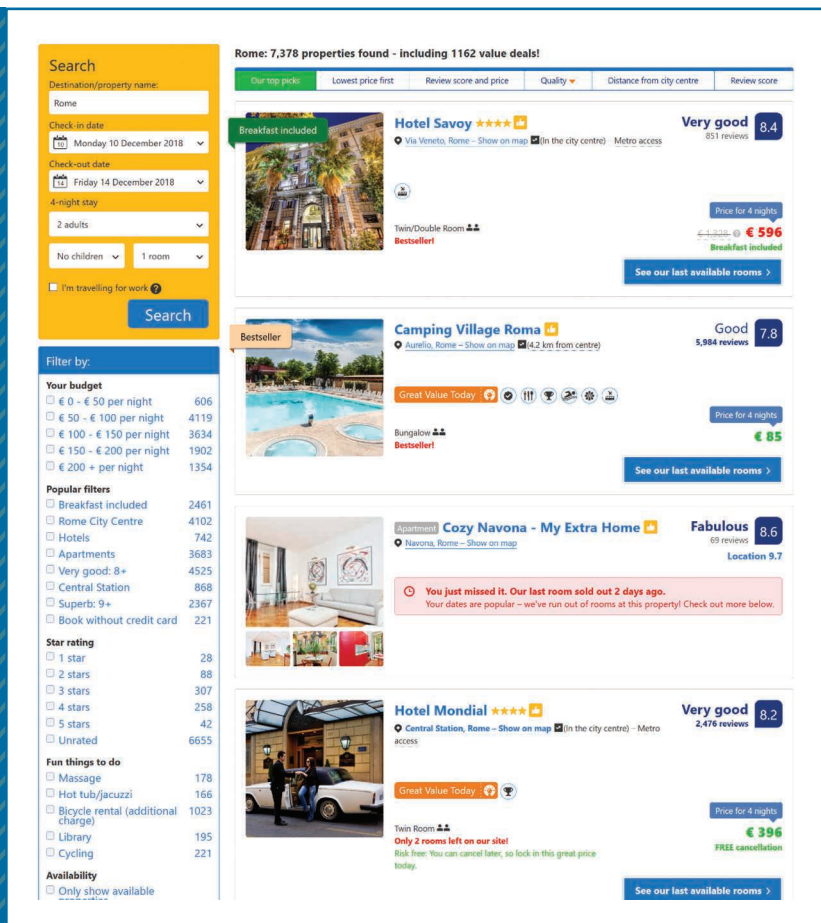


FIGURE 1. The Booking.com search interface.

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, every metamorphic test starts from an existing test case, called the *source test case*. Source test cases can be designed from scratch with any standard test-design technique, or they can be reused from an existing test suite. 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.

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. This process would include

not only the generation of inputs, but also the generation of the corresponding output assertions, truly achieving full test automation.

Approaches for the Identification of Metamorphic Relations

The effectiveness of MT is strongly influenced by the metamorphic relations used; therefore, 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

The input-driven approach involves thinking of changes to the program's inputs that should produce expected changes in the outputs.¹² The possible changes to the input parameters depend on their data types. 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 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, to identify metamorphic relations. This approach is frequently used in numerical and graph-theory programs.

Output-Driven Approach

In contrast to the previous method, the output-driven approach proposes starting from possible relations among the 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 include having a result set that 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. (The total number of publications on MT, considering all types of articles, 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

```

1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class BookingSearchTest {
5
6      @Test
7      public void searchMetamorphicTest() {
8
9          // Create query
10         BookingQueryObject query = new BookingQueryObject();
11         query.setDestination("Rome");
12         query.setCheckIn("11/12/2018");
13         query.setCheckOut("11/15/2018");
14         query.setnAdults(2);
15
16         // Source test case
17         BookingSearchResult st = Booking.search(query);
18
19         // Follow-up test case
20         BudgetFilter filter = new BudgetFilter();
21         filter.setMinBudget(120);
22         filter.setMaxBudget(180);
23         filter.setCurrency("US");
24         BookingSearchResult fut = Booking.search(query, filter);
25
26         // Metamorphic relation assertion
27         assertTrue("Not a subset", fut.isSubset(st));
28     }
29 }

```

FIGURE 2. A sample implementation of a metamorphic test in JUnit 4.

financial software, optimization programs, cybersecurity, and data analytics as well as industrial applications in organizations, such as NASA and Adobe. Only 5% of the articles 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 has been 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 (AI), including machine learning and autonomous vehicles, is worth noting. We

next highlight some successful applications of MT in the domains of compilers and AI.

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

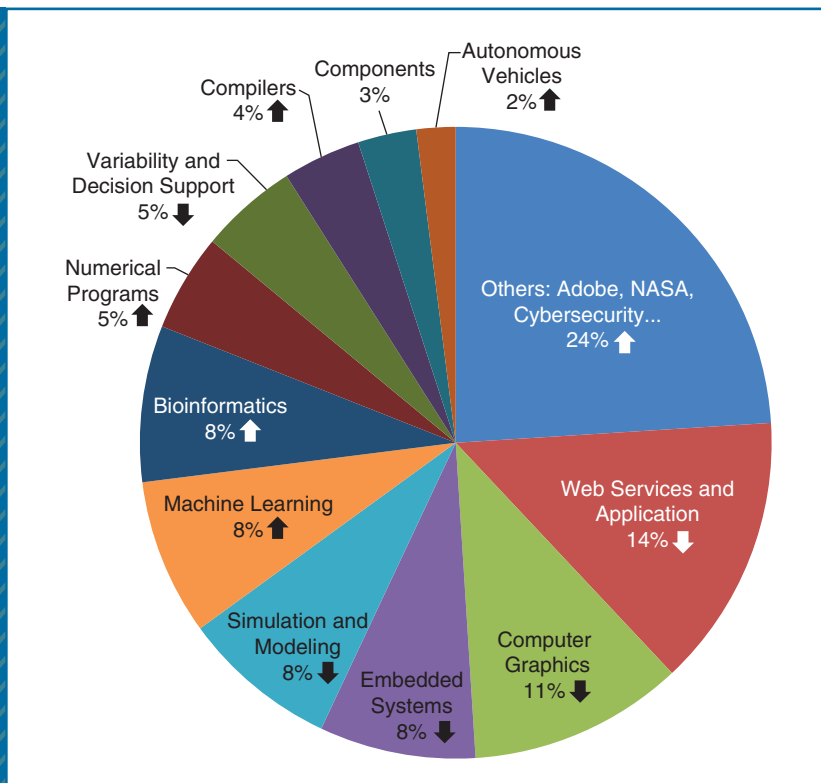


FIGURE 3. The application domains.

removing some of the dead code (statements not executed in step 1).

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 GCC and Low-Level Virtual Machine 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 follows¹⁵: 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 graphics processing unit (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 example, adding “+0.0” to an arithmetic expression). Third, an image is rendered by using the modified shader program (still through the graphics driver under test), obtaining a *variant image*. Finally, the original and variant images are compared. If the differences are significant, 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 US\$2,000. GraphicsFuzz was acquired by Google in August 2018.

AI

MT has been used for testing AI tools such as supervised and unsupervised machine-learning programs,¹⁶ which “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. MT has been used to detect real bugs in the machine-learning tools Weka and RapidMiner, among others.

MT has also proven to be effective for testing AI-driven systems. Researchers at the Fraunhofer Center for Experimental Engineering 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 in which the outcome of the drone flight should be equivalent. For example, the drone should behave consistently whether it is flying north or south, if the distances and relative positions of obstacles are the same. This approach enabled the researchers 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 were

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 the 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 Figure 4(a) shows the original trajectory of the car. The red arrow in Figure 4(b) shows the erroneous trajectory calculated by the DNN-driven system when fog was added to the scene.

Limitations

MT also has some limitations that may narrow its applicability in certain domains. First, although 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 US\$120–US\$180 per night”) are wrong (if both searches return a particular hotel in Florence,

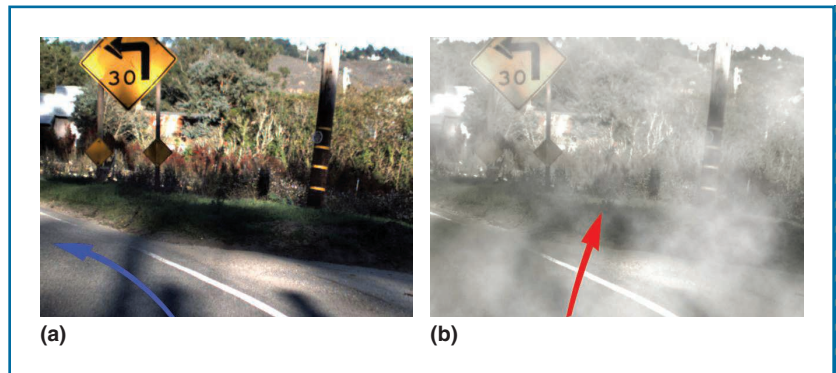



FIGURE 4. A sample of erroneous behavior found by DeepTest.¹⁹ (a) The original scene and (b) the same scene with added fog. (Source: Baishakhi Ray; used with permission.)

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 nontrivial 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.

A thriving testing technique, MT has demonstrated its 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. 

Acknowledgments

Zhi Quan Zhou acknowledges support from the Australian Research Council (project identification: LP160101691). Sergio Segura acknowledges support from the Operational Programme FEDER Andalusia and the Spanish Government under projects APOLO (US-1264651) and HORATIO (RTI2018-101204-B-C21). Dave Towey acknowledges support from the National Natural Science Foundation of China (NSFC grant 61872167).

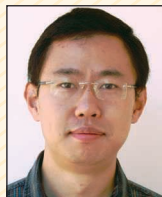
References

1. E. J. Weyuker, “On testing non-testable programs,” *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982. doi: 10.1093/comjnl/25.4.465.
2. S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortés, “A survey on metamorphic testing,” *IEEE Trans. Softw. Eng.*, vol. 42, pp. 805–824, 2016. doi: 10.1109/TSE.2016.2532875.

ABOUT THE AUTHORS



SERGIO SEGURA is an associate professor of software engineering at the University of Seville, Spain. His main research interests include test automation, search-based software engineering, and artificial intelligence-driven software engineering. Segura received a Ph.D. in computer science from the University of Seville. Contact him at sergiosegura@us.es.



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. Zhou received a Ph.D. in software engineering from The University of Hong Kong. He is one of the few earliest pioneers who opened up and established the research field of metamorphic testing. Contact him at zhiquan@uow.edu.au.



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 Ph.D. in computer science from The University of Hong Kong. He is a Member of the IEEE. Contact him at dave.towey@nottingham.edu.cn.



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

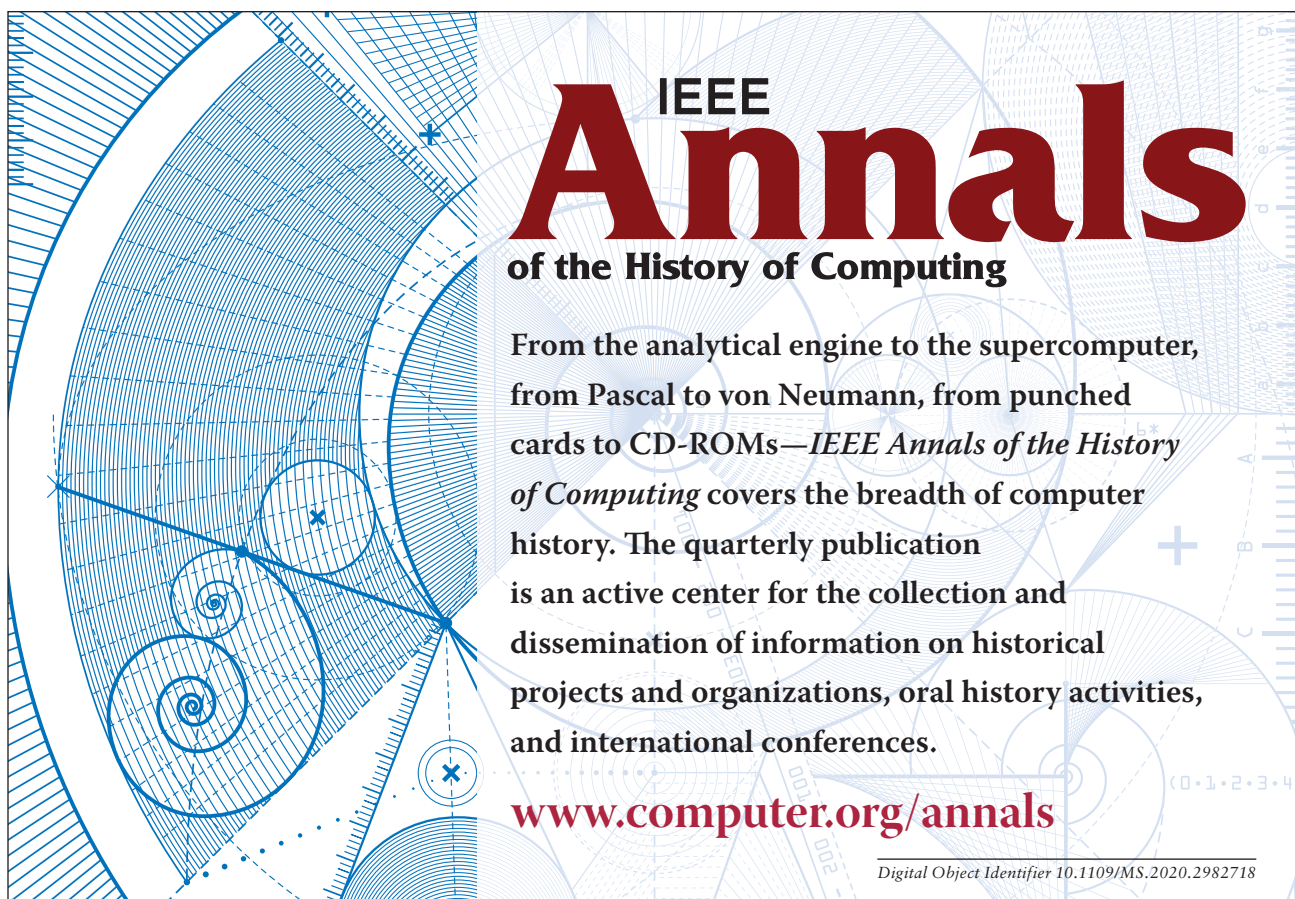
3. T. Y. Chen et al., "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys*, vol. 51, no. 1, pp. 4:1–4:27, 2018. doi: 10.1145/3143561.
4. T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Dept. Comput. Sci., Hong Kong Univ. of Science and Technology, Techn. Rep. HKUSTCS98-01, 1998.
5. Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 264–284, 2016. doi: 10.1109/TSE.2015.2478001.
6. V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2014, pp. 216–226.
7. T. Y. Chen et al., "Metamorphic testing for cybersecurity," *Comput.*, vol. 49, no. 6, pp. 48–55, 2016. doi: 10.1109/MC.2016.176.
8. M. Lindvall, D. Ganesan, R. Ardal, and R. E. Wiegand, "Metamorphic model-based testing applied on NASA DAT—An experience report," in *Proc. IEEE/ACM 37th Int. Conf. Software Engineering*, 2015, pp. 129–138.
9. S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of RESTful web APIs," *IEEE Trans. Softw. Eng.*, vol. 44, no. 11, pp. 1083–1099. doi: 10.1109/TSE.2017.2764464.
10. Google/GraphicsFuzz: A testing framework for automatically finding and simplifying bugs in graphics shader compilers, 2018. Accessed on: Jan. 24, 2020. [Online]. Available: <https://github.com/google/graphicsfuzz/>
11. S. Segura and Z. Q. Zhou, "Metamorphic testing: Introduction and applications," ACM SIGSOFT Webinar, 2017. [Online]. Available: <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," *J. Syst. Softw.*, vol. 116, pp. 177–190, 2016. doi: 10.1016/j.jss.2015.07.037.
13. H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Trans. Softw. Eng.*,

- vol. 40, no. 1, pp. 4–22, 2014. doi: 10.1109/TSE.2013.46.
14. Z. Su, “Prof. Zhendong Su.” Accessed on: June 2018. Available: <http://web.cs.ucdavis.edu/~su>
 15. A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, “Automated testing of graphics shader compilers,” in *Proc. ACM Programming Languages*, vol. 1 pp. 93:1–93:29, Oct. 2017. doi: 10.1145/3133917.
 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,” *J. Syst. Softw.*, vol. 84, pp. 544–558, 2011. doi: 10.1016/j.jss.2010.11.920.
 17. M. Lindvall, A. Porter, G. Magnusson and C. Schulze, “Metamorphic model-based testing of autonomous systems,” in *Proc. IEEE/ACM 2nd Int. Workshop Metamorphic Testing, in conjunction with the 39th Int. Conf. Software Engineering (ICSE ’17)*, 2017, pp. 35–41.
 18. Y. Tian, K. Pei, S. Jana, and B. Ray, “DeepTest: Automated testing of deep-neural-network-driven autonomous cars,” in *Proc. 40th Int. Conf. Software Engineering*, 2018, pp. 303–314.
 19. Y. Tian, K. Pei, S. Jana, and B. Ray, “DeepTest.” Accessed on: Mar. 2018. [Online]. Available: <https://deeplearningtest.github.io/deepTest>
 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 Rel.*, vol. 26, no. 3, pp. 245–269, 2016. doi: 10.1002/stvr.1594.
 21. J. Zhang et al., “Search-based inference of polynomial metamorphic relations,” in *Proc. 29th ACM/IEEE Int. Conf. Automated Software Engineering*, 2014, pp. 701–712.



IEEE COMPUTER SOCIETY
DIGITAL LIBRARY

Access all your IEEE Computer Society subscriptions at
computer.org
/mysubscriptions



IEEE Annals

of the History of Computing

From the analytical engine to the supercomputer, from Pascal to von Neumann, from punched cards to CD-ROMs—*IEEE Annals of the History of Computing* covers the breadth of computer history. The quarterly publication is an active center for the collection and dissemination of information on historical projects and organizations, oral history activities, and international conferences.

www.computer.org/annals

Digital Object Identifier 10.1109/MS.2020.2982718