Metamorphic Runtime Checking of Applications Without Test Oracles

Jonathan Bell, Columbia University
Christian Murphy, University of Pennsylvania
Gail Kaiser, Columbia University

Abstract. For some applications, it is impossible or impractical to know what the correct output should be for an arbitrary input, making testing difficult. Many machine-learning applications for “big data”, bioinformatics and cyberphysical systems fall in this scope: they do not have a test oracle. Metamorphic Testing, a simple testing technique that does not require a test oracle, has been shown to be effective for testing such applications. We present Metamorphic Runtime Checking, a novel approach that conducts metamorphic testing of both the entire application and individual functions during a program’s execution. We have applied Metamorphic Runtime Checking to 9 machine-learning applications, finding it to be on average 170% more effective than traditional metamorphic testing at only the full application level.

Introduction

During software testing, a “test oracle” [1] is required to indicate whether the output is correct for the given input. Despite a recent interest in the testing community in creating and evaluating test oracles, still there are a variety of problem domains for which a practical and complete test oracle does not exist.

Many emerging application domains fall into a category of software that Weyuker describes as “Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known [2].” Thus, in the general case, it is not possible to know the correct output in advance for arbitrary input. In other domains, such as optimization, determining whether the output is correct is at least as difficult as it is to derive the output in the first place, and creating an efficient, practical oracle may not be feasible.

Although some faults in such programs - such as those that cause the program to crash or produce results that are obviously wrong to someone who knows the domain - are easily found, and partial oracles may exist for a subset of the input domain, subtle errors in performing calculations or in adhering to specifications can be much more difficult to identify without a practical, general oracle.

Much recent research addressing the so-called “oracle problem” has focused on the use of metamorphic testing [3]. In metamorphic testing changes are made to existing test inputs in such a way (based on the program’s “metamorphic properties”) that it is possible to predict what the change to the output should be without a test oracle.

That is, if program input I produces output O, additional test inputs based on transformations of I are generated in such a manner that the change to O (if any) can be predicted. In cases where the correctness of the original output O cannot be determined, i.e., if there is no test oracle, program defects can still be detected if the new output O is not as expected when using the new input.

For a simple example of metamorphic testing (where we do have a test oracle), consider a function that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result: for instance, permuting the order of the elements should not affect the calculation, nor should multiplying each value by -1. Furthermore, other transformations should alter the output, but in a predictable way: if each value in the set were multiplied by 2, then the standard deviation should be twice that of the original set.

Through our own past investigations into metamorphic testing [4] [5] [6], we have garnered three key insights. First, the metamorphic properties of individual functions are often different than those of the application as a whole. Thus, by checking for additional and different relationships, we can reveal defects that would not be detected using only the metamorphic properties of the full application. Second, the metamorphic properties of individual functions can be checked in the course of executing metamorphic tests on the full application. This addresses the problem of generating test cases from which to derive new inputs, since we can simply use those inputs with which the functions happened to be invoked within the full application. Third, when conducting tests of individual functions within the full running application in this manner, checking the metamorphic properties of one function can sometimes detect defects in other functions, which may not have any known metamorphic properties, because the functions share application state.

Approach

In order to realize these improvements, we present a solution based on checking the metamorphic properties of the entire program and those of individual functions (methods, procedures, subroutines, etc.) as the full program runs. That is, the program under test is not treated only as a black box, but rather metamorphic testing also occurs within the program, at the function level, in the context of the running program. This will allow for the execution of more tests and also makes it possible to check for subtle faults inside the code that may not cause violations of the full program’s metamorphic properties and lead to apparently reasonable output (remember we cannot check whether that output is correct, since there is no test oracle).

In our new approach, additional metamorphic tests are logically attached to the individual functions for which metamorphic properties have been specified. Upon a function’s execution when it happens to be invoked within the full program, the corresponding function-level tests are executed as well: the arguments are modified according to the function’s metamorphic properties, the function is run again (in a sandbox, not shown) in the same program state as the original, and the output of the function with the original input is compared to that of the function with the modified input. If the result is not as expected according to the metamorphic property, then a fault has been exposed.

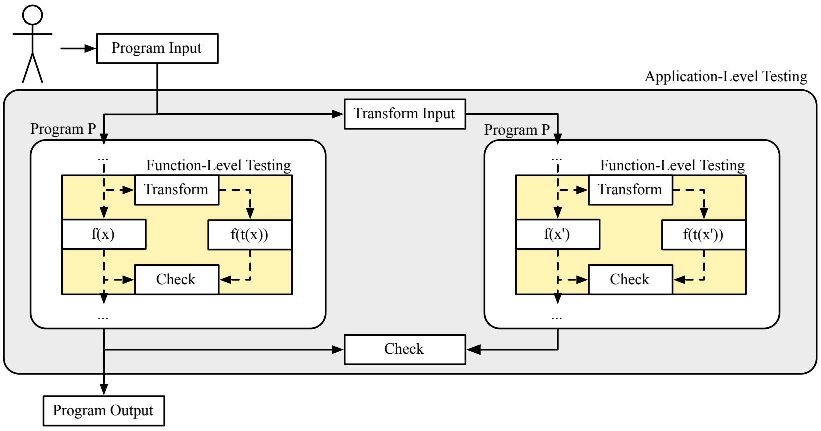

Figure 1: Model of Metamorphic Runtime Checking of program P and one of its constituent functions, f. Metamorphic Runtime Checking combines program-level metamorphic testing with function-level metamorphic checking, performing such checking automatically.

As shown in Figure 1 the tester provides a program input to a Metamorphic Runtime Checking framework, which then transforms it according to the metamorphic property of the program P (for simplicity, this diagram only shows one metamorphic property, but a program may, of course, have many). The framework then invokes P with both the original input and the transformed input; as seen at the bottom of the diagram, when each program invocation is finished, the outputs can be checked according to the property.

While each invocation of P is running, metamorphic properties of individual functions can be checked as well. As shown on the left side of Figure 1, in the invocation of P with the original program input, before a function f is called, its input x can be transformed according to one of the function's metamorphic properties to give t(x). The function is called with each input, and then f(t(x)) is evaluated according to the original value of f(x) to see if the property is violated.

Meanwhile, in the additional invocation of P (right side of the diagram), function-level metamorphic testing still occurs for function f, this time using input x', which results from the transformed program input to P. In this case, f(t(x')) and f(x') are compared.

						# of Metamorphic Properties identified at the level of:	
Application	**Domain**	**Language**	**LOC**	**Functions**	**Invariants**	**Application**	**Function**
C4.5	classification	C	5,285	141	27,603	4	40
GAFFitter	optimization	C++	1,159	19	744	2	11
JSim	simulation	Java	3,024	468	306	2	12
K-means	clustering	Java	717	46	137	4	12
LDA	topic modeling	Java	1,630	103	1,323	4	28
Lucene	information retrieval	Java	661	57	456	4	26
MartiRank	ranking	C	804	19	3,647	4	15
PAYL	anomaly detection	Java	4,199	164	19,730	2	40
SVM	classification	Java	1,213	49	2,182	4	4

Table 1: Listing of applications studied

In this example, if we used only the application-level property of P, we would run just one test. However, by also considering P's function f with one specified metamorphic property, we can now check two properties and run a total of three tests. This combined approach also allows us to reveal subtle faults at the function level that may not violate application-level properties. Our study shows that this sensitivity gain can increase the effectiveness of metamorphic testing by up to 1,350% (on average, 170%).

Evaluation

To evaluate the effectiveness of Metamorphic Runtime Checking at detecting faults in applications without test oracles, we compare it to runtime assertion checking using program invariants (a state-of-the art technique). When used in applications without test oracles, assertions can detect some programming bugs by checking that function input and output values are within a specified range, the relationships between variables are maintained, and a function's effects on the application state are as expected [7]. While satisfying the invariants does not ensure correctness, any violation of them at runtime indicates an error.

The experiments described in this section seek to answer the following research questions:

1. Is Metamorphic Runtime Checking more effective than using runtime assertion checking for detecting faults in applications without test oracles?
2. What contribution do application-level and function-level metamorphic properties make to the effectiveness of Metamorphic Runtime Checking?
3. Is Metamorphic Runtime Checking suitable for practical use?

In these experiments, we applied both runtime assertion checking and Metamorphic Runtime Checking to nine real-world applications that are representative of different domains that have no practical, general test oracles: supervised machine learning, unsupervised machine learning, data mining, discrete event simulation, and NP-hard optimization. The applications are described (along with the number of invariants, function-level and application-level properties) in Table 1.

To create the set of invariants that we could use for runtime assertion checking, we applied the Daikon invariant detector tool [8] to each application. To identify the application-level metamorphic properties for the experiment, we followed the guidelines set forth in [4], which categorizes the types of properties that applications in these domains tend to exhibit.

To identify function-level properties, we inspected the source code and hand-annotated the functions that we expected to exhibit the types of properties described in [4]. To ensure that the properties were not limited to only the ones that we could think of, some of the function-level metamorphic properties used in this experiment are based on those used in other, similar studies such as [9], [10] and [11].

Methodology

To determine the effectiveness of the testing techniques, we used mutation analysis to systematically insert faults into the source code of the applications described above, and then determined whether the mutants could be killed (i.e., whether the faults could be detected) using each approach. Mutations that yielded a fatal runtime error, an infinite loop, or an output that was clearly wrong (for instance, not conforming to the expected output syntax or simply being blank) were discarded since any reasonable approach would detect such faults.

We also did not consider "equivalent mutants" for which the inputs used in the experiment produced the same program output as the original, unmutated version, e.g., those mutants that were not on the execution path for any test case or that would not have been killed with an oracle for these inputs.

For each mutated version, we conducted runtime assertion checking with the invariants detected by Daikon. If any invariant was violated, the mutant was considered killed. We then performed Metamorphic Runtime Checking on the same mutated versions to determine whether any of the specified metamorphic properties were violated. The inputs used for mutation analysis were the same as those used for detecting invariants and verifying metamorphic properties.

Figure 2 summarize the results of our experiment evaluating the efficacy of Metamorphic Runtime Checking. Overall, Metamorphic Runtime Checking was more effective, killing 1,602 (90.4%) of the mutants in the applications, compared to just 1,498 (84.5%) for assertion checking.

Broadly speaking, Metamorphic Runtime Checking was more effective at killing mutants that related to operations on arrays, sets, collections, etc. However, further analysis could characterize the types of faults each approach is most suitable for detecting, but it follows, that runtime assertion checking and Metamorphic Runtime Checking should be used together for best results. When used in combination in our experiments, they were able to kill 95% of the mutants (totaling across all applications): only 88 of the 1,772 survived.

To understand the factors that impacted the efficacy of Metamorphic Runtime Checking, we performed a deeper analysis of the contribution of the separate mechanisms. We first determined the number of mutants killed only by application-level properties, then the number killed only by function-level properties. Table 2 shows these results.

On average, we saw a 170% improvement in the number of mutants killed when combining application-level properties with function-level properties. The variance in improvement was very large, however, showing a striking improvement of 1,350% in PAYL, while showing smaller improvement in C4.5 and MartiRank. There was no improvement at all in the JSim and LDA applications, because application-level properties had already been able to kill all mutants.

We believe that this improvement is attributed primarily to our increase in: the number of properties identified (scope); the number of tests run (scale); and the likelihood that a fault would be detected (sensitivity).

The improvement in the scope of metamorphic testing was particularly clear in the anomaly-based intrusion detection system PAYL. We were only able to identify two application-level metamorphic properties because it was not possible to create new program inputs based on modifying the values of the bytes inside the payloads, since the application only allowed for syntactically and semantically valid inputs that reflected what it considered to be "real" network traffic.

These two properties were only able to kill two of the 40 mutants. However, once we could use Metamorphic Runtime Checking to run metamorphic tests at the function level, we were able to identify many more properties that involved changing the byte arrays that were passed as function arguments, thus revealing 27 additional faults.

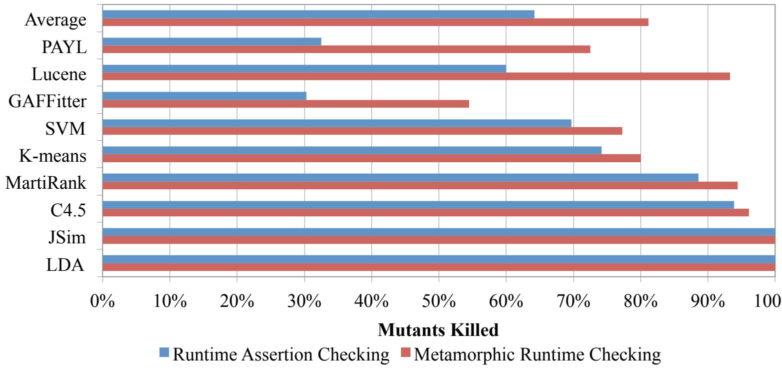

Figure 2: Results of mutation analysis comparing metamorphic runtime checking and runtime assertion checking. Metamorphic runtime checking was on average more effective.

		Mutants Killed By				
Application	**Total Mutants**	**Application-level Properties Only**	**Function-level Properties Only**	**Both Types**	**Not Killed**	**MRC % Improvement**
C4.5	856	133	37	653	33	4.71%
GAFFitter	66	2	14	20	30	63.64%
K-means	35	6	11	11	7	64.71%
JSim	36	14	0	22	0	0.00%
LDA	24	2	0	22	0	0.00%
Lucene	15	5	3	6	1	27.27%
MartiRank	413	298	22	70	23	5.98%
PAYL	40	0	27	2	11	1350.00%
SVM	287	69	23	130	65	11.56%
Average	197	59	15	104	19	169.76%

Table 2: Number of Mutants Killed by Different Types of Metamorphic Properties

Likewise, we were able to increase the scale of metamorphic testing by running many more test cases. For instance, in MartiRank, even though we specified function-level properties for only a handful of functions, many of those are called numerous times per program execution, meaning that there are many opportunities for the property to be violated.

Another reason why function-level properties were able to kill mutants not killed by application-level properties is that we were able to improve the sensitivity in terms of the ability to reveal more subtle faults, as seen in GAFFitter. In the function to calculate the "fitness" of a given candidate solution in the genetic algorithm, i.e., how close to the optimal solution (target) a candidate comes, one of the metamorphic properties is that permuting the elements in the candidate solution should not affect the result, since it is merely taking a sum of all the elements.

If, for instance, there is a mutation such that the last element is omitted from the calculation, then the metamorphic property will be violated since the return value will be different after the second function call. However, at the application level, such a fault is unlikely to be detected, since the metamorphic property simply states that the quality of the solutions should be increasing with subsequent generations. Even though the value of the fitness is incorrect, it would still be increasing (unless the omitted element had a very large effect on the result, which is unlikely), and the property would not be violated.

Performance Overhead

Although Metamorphic Runtime Checking using function-level properties is able to detect faults not found by metamorphic testing based on application-level properties alone, this runtime checking of the properties comes at a cost, particularly if the tests are run frequently. In application-level metamorphic testing, the program needs to be run one more time with the transformed input, and then each metamorphic property is checked exactly once (at the end of the program execution). In Metamorphic Runtime Checking, however, each property can be checked numerous times, depending on the number of times each function is called, and the overhead can grow to be much higher.

During the studies discussed above, we measured the performance overhead of our C and Java implementations of the Metamorphic Runtime Checking framework. Tests were conducted on a server with a quad-core 3GHz CPU running Ubuntu 7.10 with 2GB RAM. On average, the performance overhead for the Java applications was around 3.5ms per test; for C, it was only 0.4ms per test. This cost is mostly attributed to the time it takes to create sandboxes (so the side-effects of function-level metamorphic testing do not impact application-level testing).

This impact can be substantial from a percentage overhead point of view if many tests are run in a short-lived program. For instance, for C4.5, the overhead was on the order of 10x, even though in absolute terms it was well under a second. However, for most programs we investigated in our study, the overhead was typically less than a few minutes, which we consider a small price to pay for being able to detect faults in programs with no test oracle.

Future work could investigate techniques for improving the performance of a Metamorphic Runtime Checking framework. Previously we considered an approach whereby tests were only executed in application states that had not previously been encountered, and showed that performance could be improved even when the functions are invoked with new parameters up to 90% of the time [12]. It may be possible to reduce the overhead even more, for instance by running tests probabilistically (our framework already allows the tester to specify a probability for checking each function-level metamorphic property, but we turned that off for the studies presented here).

Limitations

We used Daikon to create the program invariants for runtime assertion checking. Although in practice invariants are typically generated by hand, and some researchers have questioned the usefulness of Daikon-generated invariants compared to those generated by humans [13], we chose to use the tool so that we could eliminate any human bias or human error in creating the invariants.

Additionally, others have independently shown that metamorphic properties are more effective at detecting defects than manually identified invariants [14], though for programs on a smaller scale than those in our experiment (a few hundred lines, as opposed to thousands as in many of the programs we studied).

The ability of metamorphic testing to reveal failures is clearly dependent on the selection of metamorphic properties. However, we have shown that a basic set of metamorphic properties can be used without a particularly strong understanding of the implementation - the authors knew essentially nothing about the target systems or their domains beyond textbook generality; the use of domain-specific properties from the developers of these systems might reveal even more failures [15].

Conclusion

As shown in our empirical studies, Metamorphic Runtime Checking has three distinct advantages over metamorphic testing using application-level properties alone. First, we are able to increase the scope of metamorphic testing, by identifying properties for individual functions in addition to those of the entire application. Second, we increase the scale of metamorphic testing by running more tests for a given input to the program. And third, we can increase the sensitivity of metamorphic testing by checking the properties of individual functions, making it possible to reveal subtle faults that may otherwise go unnoticed.

Acknowledgements

We would like to thank T.Y. Chen, Lori Clarke, Lee Osterweil, Sal Stolfo, and Junfeng Yang for their guidance and assistance. Sahar Hasan, Lifeng Hu, Kuang Shen, and Ian Vo contributed to the implementation of the Metamorphic Runtime Checking framework.

Bell and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CCF-1302269, NSF CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

ABOUT THE AUTHORS

Jonathan Bell is a Ph.D. student in Software Engineering at Columbia University. His research interests include software testing, program analysis, and fault reproduction. He's received an M Phil, MS and BS in Computer Science from Columbia University.

Dept. of Computer Science
Columbia University
New York, NY 10027
Phone: 212-939-7184
E-mail: jbell@cs.columbia.edu

Christian Murphy is an Associate Professor of Practice and Director of the Master of Computer and Information Technology program at The University of Pennsylvania. His primary interests are software engineering, systems programming, and mobile/embedded computing. He received his Ph.D. in Computer Science from Columbia University.

Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Phone: 215-898-0382
E-mail: cdmurphy@cis.upenn.edu

Gail E. Kaiser is a Professor of Computer Science at Columbia University and a Senior Member of IEEE. Her research interests include software reliability and robustness, information management, social software engineering, and software development environments and tools. She has served as a founding associate editor of ACM TOSEM and as an editorial board member for IEEE Internet Computing. She received her Ph.D. and MS from CMU and her ScB from MIT.

Dept. of Computer Science
Columbia University
New York, NY 10027
Phone: 212-939-7184
E-mail: kaiser@cs.columbia.edu

REFERENCES

1. Pezzé, M. and M. Young, Software Testing and Analysis: Process, Principles and Techniques. 2007: Wiley.
2. Weyuker, E.J., On testing non-testable programs. Computer Journal, 1982. 25(4): p. 465-470.
3. Chen, T.Y., S.C. Cheung, and S.M. Yiu, Metamorphic testing: a new approach for generating next test cases. 1998, Dept. of Computer Science, Hong Kong Univ. of Science and Technology.
4. Murphy, C., et al., Properties of Machine Learning Applications for Use in Metamorphic Testing, in Proc. of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE). 2008. p. 867-872.
5. Murphy, C., et al., On Effective Testing of Health Care Simulation Software, in Proc. of the 3rd International Workshop on Software Engineering in Health Care. 2011.
6. Murphy, C., K. Shen, and G. Kaiser, Automated System Testing of Programs without Test Oracles, in Proc. of the 2009 ACM International Conference on Software Testing and Analysis (ISSTA). 2009. p. 189-199.
7. Nimmer, J.W. and M.D. Ernst, Automatic generation of program specifications, in Proc. of the 2002 International Symposium on Software Testing and Analysis (ISSTA). 2002. p. 232-242.
8. Ernst, M.D., et al., Dynamically discovering likely programming invariants to support program evolution, in Proc. of the 21st International Conference on Software Engineering (ICSE). 1999. p. 213-224.
9. Barus, A.C., et al., Testing of Heuristic Methods: A Case Study of Greedy Algorithm. Lecture Notes in Computer Science, 2011. 4890: p. 246-260.
10. Kanewala, U. and J.M. Bieman, Techniques for Testing Scientific Programs Without an Oracle, in Proc. of the 2013 International Workshop on Software Engineering for Computational Science and Engineering. 2013.
11. Cheatham, T.J., J.P. Yoo, and N.J. Wahl, Software testing: a machine learning experiment, in Proc. of the ACM 23rd Annual Conference on Computer Science. 1995. p. 135-141.
12. Murphy, C., et al., Automatic Detection of Previously-Unseen Application States for Deployment Environment Testing and Analysis, in Proc. of the 5th International Workshop on Automation of Software Test (AST). 2010.
13. Polikarpova, N., I. Ciupa, and B. Meyer, A comparative study of programmer-written and automatically inferred contracts, in Proc. of the 2009 International Symposium on Software Testing and Analysis (ISSTA). 2009. p. 93-104.
14. Hu, P., et al., An empirical comparison between direct and indirect test result checking approaches, in Proc. of the 3rd International Workshop on Software Quality Assurance. 2006. p. 6-13.
15. Xie, X., et al., Application of Metamorphic Testing to Supervised Classifiers, in Proc. of the 9th International Conference on Quality Software (QSIC). 2009. p. 135-144.