A NEW APPROACH FOR AN INTEGRATED TOOL USING OPTIMIZED METAMORPHIC RELATIONS

[1]CHITTINENI ARUNA, [2]R.SIVA RAM PRASAD

[1]Research Scholar, Acharya Nagarjuna University, Assistant Professor, Dept of CSE, KKR & KSR Institute of Technology and Sciences, Guntur Dist, Andhra Pradesh, India.
[2] Research Guide, Dept. of CSE and Head, Dept. of IBS, Acharya Nagarjuna University, Nagarjuna Nagar, Guntur Dist, Andhra Pradesh, India.
Email: chittineni.aruna@gmail.com,

Aim: Identification of constraint expressions that reveal failures support automatic debugging. Whenever we are using these relations on our integrated method it will give better performance gain compared to existing approaches. And also increase the performance of generating test cases.

Abstract: We present a semi-proving method for verifying necessary conditions for program correctness. Previously an integrated program method was introduced for Testing, Debugging and Proving a program results for check that program may have any modification in execution. Previous method can be developed based on metamorphic relations, Symbolic analysis of and path constraint simplifications. Metamorphic method can be applied to correctness of a program to tested inputs related to untested inputs. These Methods can be applied for generating test cases with original tested inputs. Prior approaches use four metamorphic relations to initiate metamorphic testing for the verification of software output without a complete testing oracle. So the main disadvantage of this aspect is shortage of test case generation. To increase the frequency of number of test cases we propose a new approach. Our proposed new approach contains all the metamorphic relations available in testing process for generating the test cases.

Index Terms: Symbolic analysis, metamorphic relations, testing debugging global symbolic evaluation, program proving.

I. INTRODUCTION

Program proving and program testing are the two means of verifying the correctness of a program. The former uses a mathematical proof to show that the program in question is correct. It is, however, not popular in the industry because of the problems of automation and the complexity of proofs even for small programs. The latter, program testing, has remained the major means of establishing confidence in software correctness. Testing can reveal a fault only when the execution of the corresponding faulty element causes a failure. A common approach for testing numerical and scientific programs is to check whether they preserve some identity relations exhibited by the functions in question. The notion of data diversity is based on a very novel idea aimed at overcoming the problems associated with N-version programming. It has been developed from the perspective of fault tolerance rather than fault detection.
More recently, a metamorphic testing method was proposed by Chen et al. It has been proposed as a property-based test case selection strategy. It is based on the intuition that even if no failure is revealed by a test case selected according to some strategies, it still has useful information. Thus, follow-up test cases should be further constructed from the original test cases with reference to some necessary conditions of the problem to be implemented. Such necessary properties guiding the construction of follow-up test cases are known as metamorphic relations.

There are fundamental differences between metamorphic testing and the other methods outlined above, even though all of them propose to test programs against selected properties of implemented

(a) In metamorphic testing, the properties are not limited to identity relations. It has been applied, for instance, to verify the convergence of solutions of partial differential equations with respect to the refinement of grid points. On the other hand, apart from a couple of examples on error bounds given by Weyuker, all the other techniques have only made use of identity relations.

(b) When compared with data diversity, a further difference is that other test cases used in data diversity are basically expressed forms of the original test cases. This constraint is necessary because the technique is applied in fault tolerance, with the objective of applying alternate ways to process the original test case but using the same program. In metamorphic testing, although other test cases are also derived from the original test cases, they are not limited by this constraint. Having said that, it should be pointed out that metamorphic testing and Cody and Waite's technique do not differ in this aspect.

(c) For program checkers, the main objective is to provide a probabilistic oracle for any given test case, so as to estimate whether the corresponding output is likely to be correct. The generation of additional test cases is only a by-product of the system.

Metamorphic testing, however, does not aim at providing an alternative oracle for a given input. Instead, it postulates that even though we do not know whether a test case is successful in the absence of an oracle, its result may still carry very useful information: We can run a follow-up test case and compare the series of results against selected properties of the implemented functions.

Program proving suffers from the complexity of the proofs and the problems in automation even for relatively simple programs. Limitation of program testing is the oracle problem. An oracle is a mechanism against which testers can decide whether the outcome of the execution of a test case is correct. An ideal oracle can "provide an unerring pass/fail judgment". And obtaining such an automated oracle is a complicated task. Debugging is based on the novel idea of running the same program on re-expressed forms of the original input to avoid the high cost of developing multiple versions in N-version programming. It was proposed from the perspective of fault tolerance rather than fault detection, and since then has only been advocated as a fault tolerance technique.

Our approach proposes an online testing approach for testing services. The main contributions of the preliminary version include:

1. It proposes to apply the notion of metamorphic testing to services computing to alleviate the test oracle problem. It constructs more test cases to reveal faults than those ordinarily required when test oracles are known.

2. It proposes to realize the metamorphic testing mechanism as a metamorphic service in services computing that encapsulates a service under, executes test cases and cross-validates their test results. Such realization integrates seamlessly to the existing SOA framework. It automates the construction of follow-up test cases and their test results checking.
The main extended contributions of our approach include:

3. It devises the use of the successful test cases for offline testing as the original test cases for online testing. Since in an offline testing, the environment of services can be controlled by testers, test oracle could be defined. Thus, any failure revealed by our metamorphic testing approach will be due to the failures in the online testing mode.

II. METHODS

We review related work that uses machine learning approaches as pseudo-oracles, as well as related work on metamorphic testing and other approaches to alleviating the test oracle problem. For brevity, we shall focus on the testing of software with graphical interfaces. The test oracle problem has also been studied in other contexts. Ostrand et al. propose an integrated environment so that testers can easily review and modify their test scripts. Dillon and Ramakrishna prune the search space of test oracles constructed from a specification.

2.1 Machine Learning Fundamentals

In general, input to a supervised machine learning application consists of a set of training data that can be represented by two vectors of size k. One vector is for the k training samples S = <s0, s1... sk-1> and the other is for the corresponding class labels C = <c0, c1... ck-1>. Each sample s 2 S is a vector of size m, which represents m features from which to learn. Each label ci in C is an element of a finite set of class labels, that is, c 2 L = fl0, l1, ln-1g, where n is the number of possible class labels.

Supervised ML applications consist of two phases. The first phase (called the training phase) analyzes the training data; the result of this analysis is a model that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the testing phase), the model is applied to another, previously-unseen data set (the testing data) where the labels are unknown.

For services, as we will discuss, formal test oracle may be unavailable, however. The expected behavior of a service that represents business goods and services changes according to the environment. Such an expected behavior is relative to the behaviors of competing services or other services.

Testing

In software testing, **test automation** is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes to predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or add additional testing that would be difficult to perform manually.

Debugging:

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

Proving

Using this method, program states collected from executions of concrete test cases are generalized by means of abstractions. Then, a theorem prover will check the generalized set of states against a coverage

criterion and against certain safety properties. When the check is successful, the safety properties are proved.

Verifying metamorphic relations and verifying safety properties, therefore, are complementary to each other — this has been discussed previously. Secondly, the objectives of the two methods are different. Yorsh et al's method "is oriented towards finding a proof rather than detecting real errors," and "does not distinguish between a false error and a real error."

III. EXISTING APPROACH

Program proving suffers from the complexity of the proofs and the problems in automation even for relatively simple programs. Limitation of program testing is the oracle problem. An oracle is a mechanism against which testers can decide whether the outcome of the execution of a test case is correct. An ideal oracle can "provide an unerring pass/fail judgment". And obtaining such an automated oracle is a complicated task. Debugging is based on the novel idea of running the same program on re-expressed forms of the original input to avoid the high cost of developing multiple versions in N-version programming. It was proposed from the perspective of fault tolerance rather than fault detection, and since then has only been advocated as a fault tolerance technique. Consequently, properties used in data diversity are intrinsically limited to identity relations. When an identity relation in data diversity/program checker/self-tester has been violated, the failed execution can be analyzed automatically to reveal more information about the failure. This separate entity approach is both time consuming and laborious process. So a better and automated system is required that can address these issues.

We address the above specified problems by means of a semi-proving method, which integrates program proving, testing, and debugging. The cycle is automated without the need for manual loading of the program into different tools to attain their respective objectives. Sophisticated techniques and automated tools for symbolic analysis and path constraint simplifications have been developed to facilitate more effective parallelism and optimization of programs between the three different stages of SDLC. Instead of employing any oracles for initiating testing we propose to implement metamorphic testing. Metamorphic testing is a technique for the verification of software output without a complete testing oracle.

The subject program is verified through metamorphic relations (MR). It is unlikely for a single MR to detect all possible faults. Therefore, four MRs that are quite different from one another with a view to detecting various faults were used here. Finding good MRs requires knowledge of the problem domain, understanding of user requirements, as well as some creativity. These MRs are identified according to equivalence and nonequivalence relations among regular expressions. So this kind of testing facilitates in an automated addressing of all possible forms of failures . Symbolic analysis and path constraint simplifications along with Metamorphic testing helps to integrate the three different stages of SDLC thus enabling an automated development environment that is both robust and fast.

IV. OUR PROPSED APPROACH

Our approach is based on the concept of metamorphic testing. To facilitate that approach, we must identify the relations that the algorithms are expected to exhibit between sets of inputs and sets of outputs. Once those relations have been determined, we then analyze the algorithms to decide whether the relations are necessary properties to indicate correctness during testing; that is to say, if the implementation does not exhibit that property, then there is a defect. If the relation is not a necessary property, it can still be used for the purpose of validation, that is, whether the algorithm satisfies the requirement.

4.1. Metamorphic Testing

One popular technique for testing programs without a test oracle is to use a "pseudo-oracle" (Davis and Weyuker, 1981), in which multiple implementations of an algorithm process the same input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes (Knight and Leveson, 1986) . However, even without multiple implementations, often these applications exhibit properties such that if the input were modified in a certain way, it should be possible to predict the new output, given the original output. This approach is known as metamorphic testing. Metamorphic testing can be implemented very easily in practice. The first step is to identify a set of properties ("metamorphic relations", or MRs) that relate multiple pairs of inputs and outputs of the target program. Then, pairs of source test cases and their corresponding follow-up test cases are constructed based on these MRs. We then execute all these test cases using the target program, and check whether the outputs of the source and follow-up test cases satisfy their corresponding MRs. A simple example of a function to which metamorphic testing could be applied would be one 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 would multiply each value by -1, since the deviation from the mean would still be the same.

Furthermore, there are other transformations that will alter the output, but in a predictable way. For instance, if each value in the set is multiplied by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers (the source test cases), we can use these metamorphic relations to create three more sets of follow-up test cases (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2); moreover, given the result of only the source test case, we can predict the others.

It is not hard to see that metamorphic testing is simple to implement, effective, easily automatable, and independent of any particular programming language. Further, for the identification of the MRs, which is the most crucial step in metamorphic testing, there are several principles of both white-box and black-box can be followed, such as, logical hierarchy, difference in execution traces, user's pro files, etc (Chen et al., 2004). For example, based on the principle of "difference in execution traces", we intend to select MRs with more differences between the execution traces of source test cases and follow-up test cases. Here is an illustration: the Shortest-Path program SP accepts 3 parameters as inputs: a given graph G, a starting node s, and an ending node e. SP (G, s, e) return the shortest path between s and e. Let us Consider the two following MRs as examples:

MR1: j SP (G, s, e)j = j SP(G, s, m)j + j SP(G, m, e)j, where m denotes a visited node between s and e returned by SP(G, s, e). And MR2: j SP (G, s, e)j = j SP(G, e, s)j. Apparently, these two MRs will execute different path-pairs (source path and follow-up path), and a path pair with more difference is preferred as a better MR. of course in order to decide which MR will result in more execution difference, we can just run the program and collect the real coverage information. But we also can acquire this information simply by analyzing the mechanism of the algorithm, without any execution. Suppose the algorithm is a forward-search algorithm, MR2 is likely to lead to more execution difference. However if the algorithm is a 2-way search method, MR2 will not necessarily yield more execution difference. Apart from the above principles, we can also harness the domain knowledge. This is a useful feature since in scientific computing the programmer may, in fact, also be the domain expert and will know what properties of the program will be used more heavily or are more critical. Perhaps more importantly, it is clear that metamorphic testing can be very useful in the absence of a test oracle, that is, when the correct output cannot be verified: regardless of the input values, if the metamorphic relations are violated, then there is likely a defect in the implementation.

4.2. Metamorphic Relations

In previous work (Murphy et al., 2008), we broadly classified Six types of metamorphic relations (MRs) applicable in general to many different types of machine learning applications, including both supervised and unsupervised ML.

In this work, however, our approach calls for focusing on the specific metamorphic relations of the application under test; we would expect that we could then create more follow-up test cases and conceivably reveal more defects than by using more general MRs. In particular, we define the MRs that we anticipate classification algorithms to exhibit, and define them more formally as follows.

MR-0: Consistence with affine transformation: The result should be the same if we apply the same arbitrary affine transformation function, $f(x) = kx + b$, (k 6= 0) to every value x to any subset of features in the training data set S and the test case ts.

MR-1.1: Permutation of class labels: Assume that we have a class-label permutation function Perm () to perform one-to-one mapping between a class label in the set of labels L to another label in L. If the source case result is li, applying the permutation function to the set of corresponding class labels C for the follow-up case, the result of the follow-up case should be Perm (li).

MR-1.2: Permutation of the attribute: If we permute the m attributes of all the samples and the test data, the result should remain unchanged.

MR-2.1: Addition of uninformative attributes: An uninformative attribute is one that is equally associated with each class label. For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we add an uninformative attribute to S and respectively a new attribute in st. The choice of the actual value to be added here is not important as this attribute is equally associated with the class labels. The output of the follow-up test case should still be li

MR-2.2: Addition of informative attributes: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we add an informative attribute to S and ts such that this attribute is strongly associated with class li and equally associated with all other classes. The output of the follow-up test case should still be li.

MR-3.1: Consistence with re-prediction: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we can append ts and ct to the end of S and C respectively. We call the new training dataset S' and C'. We take S', C' and ts as the input of the follow-up case, and the output should still be li.

MR-3.2: Additional training sample: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we duplicate all samples in S and L which have label li. The output of the follow-up test case should still be li.

MR-4.1: Addition of classes by duplicating samples: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we duplicate all samples in S and C that do not have label li and concatenate an arbitrary symbol "*" to the class labels of the duplicated samples. That is, if the original training set S is associated with class labels <A, B, C> and li is A, the set of classes in S in the follow-up input could be <A, B, C, B*, C*>. The output of the follow-up test case should still be li.

Another derivative of this metamorphic relation is that duplicating all samples from any number of classes which do not have label li will not change the result of the output of the follow-up test case.

MR-4.2: Addition of classes by re-labeling samples: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we reliable some of the samples in S and C which have label other than li and concatenate an arbitrary symbol "*" to their class labels. That is, if the original training set S is associated with class labels <A, B, B, B, C, C, C> and c0 is A, the set of classes in S in the follow-up input may become <A, B, B, B*, C, C*, C*>. The output of the follow-up test case should still be li.

MR-5.1: Removal of classes: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we remove one entire class of samples in S of which the label is not li. That is, if the original training set S is associated with class labels <A, A, B, B, C, C> and li is A, the set of classes in S in the follow-up input may become <A, A, B, B>. The output of the follow-up test case should still be li.

MR-5.2: Removal of samples: For the source input, suppose we get the result ct = li for the test case ts. In the follow-up input, we remove part of some of the samples in S and C of which the label is not li . That is, if the original training set S is associated with class labels <A, A, B, B, C, C> and li is A, the set of classes in S in the follow-up input may become <A, A, B, C>. The output of the follow-up test case should still be li.

V. EMPIRICAL RESULTS

In our approach we will describe the empirical results with Testing, debugging and proving compared with previous metamorphic relations. In our previous Integrated Approach we will use only four metamorphic relations for generating test cases results. Due to this approach we improve the performance of debugging commands for verifying all the related concepts in the integrated tool.

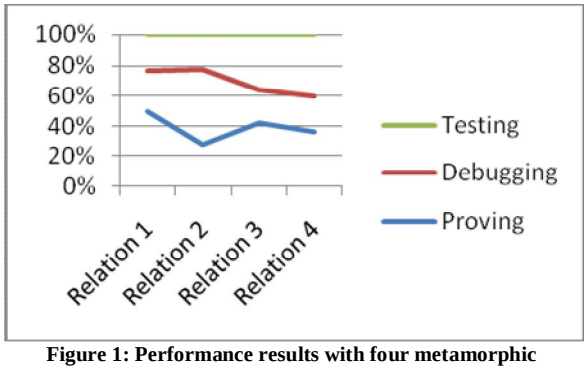


Figure 1: Performance results with four metamorphic relations.

Once those relations have been determined, we then analyze the algorithms to decide whether the relations are necessary properties to indicate correctness during testing; that is to say, if the implementation does not exhibit that property, then there is a defect. If the relation is not a necessary property, it can still be used for the purpose of validation, that is, whether the algorithm satisfies the requirement.

Using above relations presented in the Metamorphic Testing process we will improve the relations according to the testing results. Because these results are compared to previous results of the test cases generations. In our approach we will introduce the all the metamorphic relations presented in the segment process.

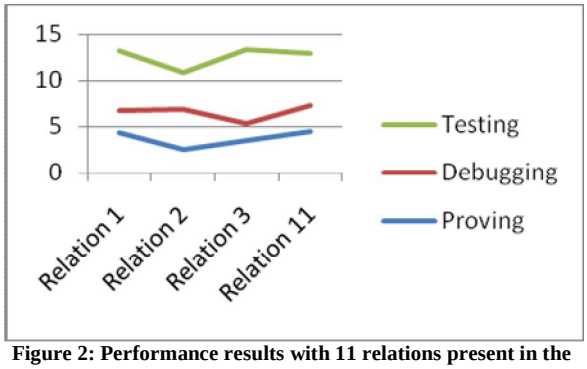


Figure 2: Performance results with 11 relations present in the metamorphic testing.

In the above discussion we will concentrate on testing only. Because in the presence of previous Metamorphic relations purely concentrate on the proving of the program generation and debugging commands but they are not use for test cases with equal generation of related concepts. But it is not

sufficient for detecting malicious exceptions in the program testing process. So we will improve the frequency of the test cases for detecting malicious exceptions with generating the test cases efficiently.

CONCLUSION

In our approach we will describe the following things: Program proving suffers from the complexity of the proofs and the problems in automation even for relatively simple programs. Debugging is based on the novel idea of running the same program on re-expressed forms of the original input to avoid the high cost of developing multiple versions in N-version programming this separate entity approach is both time consuming and laborious process. So a better and automated system is required that can address these issues. Symbolic analysis and path constraint simplifications along with Metamorphic testing helps to integrate the three different stages of SDLC thus enabling an automated development environment that is both robust and fast. We propose to highlight all the available metamorphic relations (around 11) applicable to a source code and dynamically apply those that are most suited and relevant to the code for generating test cases. The system is flexible enough to initiate testing procedures into multiple modules of prevalent code . End result is an automated development environment that is more robust and fast.

REFERENCES

[1] H. Agrawal, J.R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," Proceedings of the 6th International Symposium on Software Reliability Engineering (ISSRE '95), pp. 143–151. Los Alamitos, CA: IEEE Computer Society Press, 1995

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding bugs in dynamic web applications," Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), pp. 261–272. New York, NY: ACM Press, 2008.

[3] W.K. Chan, S.C. Cheung, and K.R.P.H. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," International Journal of Web Services Research, vol. 4, no. 2, pp. 60–80, 2007.

[4] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen, "In black and white: an integrated approach to class-level testing of object oriented programs," ACM Transactions on Software Engineering and Methodology, vol. 7, no. 3, pp. 250–295, 1998.

[5] H.Y. Chen, T.H. Tse, and T.Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels," ACM Transactions on Software Engineering and Methodology, vol. 10, no. 1, pp. 56–109, 2001.

[6] T.Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[7] T.Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002). IEEE Computer Society Press, Los Alamitos, California, 2002.

[8] Chen, T.Y., Cheung, S.C., and Yiu, S.M. Metamorphic testing: a new approach for generating next test cases, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, '1998'.

[9] Chen, T.Y., Feng, J., and Tse, T.H. Metamorphic testing of programs on partial differential equations: a case study, In Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), IEEE Computer Society Press, Los Alamitos, California, '2002'

[10] H. He and N. Gupta, "Automated debugging using path based weakest preconditions," Fundamental Approaches to Software Engineering (FASE 2004), Lecture Notes in Computer Science, vol. 2984, pp. 267–280. Berlin, Germany: Springer, 2004.

[11] H. He and N. Gupta, "Automated debugging using path based weakest preconditions," Fundamental Approaches to Software Engineering (FASE 2004), Lecture Notes in Computer Science, vol. 2984, pp. 267–280. Berlin, Germany: Springer, 2004.

[12] M. Pezz`e and M. Young, Software Testing and Analysis: Process, Principles, and Techniques. New York, NY: Wiley, 2008.

[13] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!," Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), pp. 145–156. New York, NY: ACM, 2006.

★★★