Optimized Metamorphic Relations for an automated SDLC

Chillakanti Rambabu [1], M.R.RajaRamesh [2]

[1]Student, Sri Vasavi Engineering College, TadepalliGudem, Andhra Pradesh

[2]Assosiate Professor, Sri Vasavi Engineering College ,TadepalliGudem, Andhra Pradesh

Abstract: Previously an integrated method for program proving, testing, and debugging is developed using the concept of metamorphic relations, symbolic analysis and path constraint simplifications. Metamorphic testing observes that even if the executions do not result in failures, they still bear useful information. That method extrapolates from the correctness of a program for tested inputs to the correctness of the program for related untested inputs. Follow-up test cases should be constructed from the original set of test cases with reference to selected necessary properties of the specified function. Such necessary properties of the function are called metamorphic relations. Prior approaches use four metamorphic relations to initiate metamorphic testing for the verification of software output without a complete testing oracle. To increase the frequency of number of test cases we propose to highlight all the available metamorphic relations applicable to a source code and dynamically apply those that are most suited and relevant to the code for generating test cases. Identification of constraint expressions that reveal failures support automatic debugging. The extra metamorphic relation drives the generation and initiation of more testing procedures into various modules of the prevalent code resulting in a better performance gain.

IndexTerms: Software/program verification, symbolic execution, testing and debugging.

I. INTRODUCTION

The relationship between testing and debugging is an intimate one. Thorough testing requires an under-standing not only of program requirements but also of the program implementation. To understand a pro-gram's implementation the program's semantics and syntax must be understood. The tester, often the author of the program, exploits this understanding to design tests which are effective in eliciting program failures. Once the program fails, various debugging techniques and tools are employed to locate the bug. In this paper we describe a practical slicing tool for java language programs.

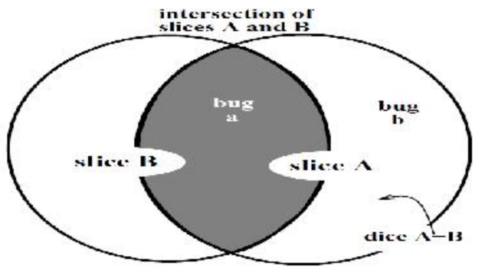

Fig 1: Bugs and slices interaction

Dynamic test-generation tools, such as DART, Cute, and EXE, find failures by executing an application on concrete input values, and then creating additional input values by solving symbolic constraints derived from exercised control flow paths. To date, such approaches have not been practical in the domain of Web applications.

The output of a Web application is typically an HTML page that can be displayed in a browser. Our goal is to find faults that are manifested as Web application crashes or as malformed HTML. Some faults may terminate the application, such as when a Web application calls an undefined function or reads a nonexistent file. Previously an integrated method for program proving, testing, and debugging is developed using the concept of metamorphic relations, symbolic analysis and path constraint simplifications. Metamorphic testing observes that even if the executions do not result in failures, they still bear useful information. That method extrapolates from the correctness of a program for tested inputs to the correctness of the program for related untested inputs.

Metamorphic testing observes that even if the executions do not result in failures, they still bear useful information. Follow-up test cases should be constructed from the original set of test cases with reference to selected necessary properties of the specified function. Such necessary properties of the function are called metamorphic relations. 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.

II. RELATED WORK

A test oracle is a mechanism that reliably decides whether a test succeeds. 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. Intuitively, it is hard to define the expected behavior explicitly in the first place. Tsai et al. (2004), for example, suggest using a progressive ranking of similar implementations of a service description to alleviate the test oracle problem. The behaviors of different implementations of the same service vary in general. Test results of a particular group of implementations cannot reliably be served as the expected behavior of a particular implementation of the same service on the same test case. Also, a typical SOA application may comprise collaborative services of multiple organizations, knowing all the implementations is impractical.

This paper extends the preliminary version to propose an online testing approach for testing services. The main contributions of the preliminary version include:

a) 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.
b) 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.

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.

There are two general approaches to test automation:

- **Code-driven testing**. The public (usually) interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.
- **Graphical user interface testing**. A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

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 behaves as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another. Many books have been written about debugging (see below: Further reading), as it involves numerous aspects, including interactive debugging, control flow, integration testing, log files, monitoring (application, system), memory dumps, profiling, Statistical Process Control, and special design tactics to improve detection while simplifying changes.

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.

The properties of interest, however, are very different between the two approaches. Yorsh et al.'s method verifies safety properties such as the absence of memory leaks and the absence of null pointer dereference errors. On the other hand, semi-proving is interested in metamorphic relations, which are more relevant to *logic errors*. Logic errors seldom cause memory abnormalities like memory access violations, segmentation faults, or memory leaks. Instead, they produce incorrect outputs. Furthermore, the safety properties discussed by Yorsh et al. [57] are at the coding level, but metamorphic relations are usually identified from the problem domain.

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 SYSTEM

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 oracle's 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. PROPOSED METHODOLOGY

Uses an integrated method that covers proving, testing, and debugging. This automated cycle of SDLC is obtained by using path constraint simplifications along with metamorphic testing. Prior approaches use four metamorphic relations to initiate Metamorphic testing for the verification of software output without a complete testing oracle. The system is constrained with the use of only four metamorphic relations. This constrained system produces constrained number of test cases. 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.

V. PERFORMANCE ANALYSIS

Our approach is based on the concept of metamorphic testing (Chen et al., 1998), summarized below. 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.

5.1 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

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 it's 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 relabel 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.

VI. CONCLUSION

In this paper 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.

VII. 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] M. Blum, M. Luby, and R. Rubinfeld, “Selftesting / correcting with applications to numerical problems,” Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC ’90), pp. 73–83. New York, NY: ACM Press, 1990. Also Journal of Computer and System Sciences, vol. 47, no. 3, pp. 549–595, 1993.

[4] 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.

[5] T.Y. Chen, J. Feng, and T.H. Tse, “Metamorphic testing of programs on partial differential equations: a case study,” Proceedings of the 26th Annual International Computer Software and Applications Conference (COMP- SAC 2002), pp. 327–333. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[6] T.Y. Chen, T.H. Tse, and Z.Q. Zhou, “Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing,” Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), ACM SIGSOFT Software Engineering Notes, vol. 27, no. 4, pp. 191–195, 2002.

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