Dynamic Inference of Likely Metamorphic Properties to Support Differential Testing

Fang-Hsiang Su*, Jonathan Bell*, Christian Murphy† and Gail Kaiser*
*Department of Computer Science, Columbia University, New York, NY USA
Email: {mikefhsu,jbell,kaiser}@cs.columbia.edu
†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA USA
Email: cdmurphy@cis.upenn.edu

***Abstract*—Metamorphic testing is an advanced technique to test programs without a true test oracle such as machine learning applications. Because these programs have no general oracle to identify their correctness, traditional testing techniques such as unit testing may not be helpful for developers to detect potential bugs. This paper presents a novel system, KABU, which can dynamically infer properties of methods' states in programs that describe the characteristics of a method before and after transforming its input. These Metamorphic Properties (MPs) are pivotal to detecting potential bugs in programs without test oracles, but most previous work relies solely on human effort to identify them and only considers MPs between input parameters and output result (return value) of a program or method. This paper also proposes a testing concept, *Metamorphic Differential Testing* (MDT). By detecting different sets of MPs between different versions for the same method, KABU reports potential bugs for human review. We have performed a preliminary evaluation of KABU by comparing the MPs detected by humans with the MPs detected by KABU. Our preliminary results are promising: KABU can find more MPs than human developers, and MDT is effective at detecting function changes in methods.**

I. INTRODUCTION

Metamorphic testing [3] is a technique for testing programs that might traditionally be called "non-testable" [5]. Such programs lack a "true" test oracle [8] — they are programs for which we are unable to determine a priori what the output result should be for every input, so traditional test cases cannot be constructed. In metamorphic testing, we compare the outputs of methods across different inputs, rather than directly comparing inputs and outputs. Metamorphic testing has been applied to diverse fields such as machine learning [12], web services [4] and simulation [13], and has been shown to have similar fault-detection capabilities to traditional testing with oracles [10].

A pervasive problem in metamorphic testing is the identification of Metamorphic Properties (MPs). Typically, human testers are tasked with annotating the system under test with MPs that they believe should apply. These properties can be at the level of granularity of individual methods, or at the coarser granularity of system-level properties. Our prior work shows that metamorphic testing is most effective when several different levels of granularity are considered simultaneously [11]. In this case, it can require substantial time and effort on the part of testers to identify all of these MPs.

We present KABU, a novel approach that guides developers to likely MPs that may apply to their systems. Our approach to dynamically inferring likely MPs is inspired by previous work on inferring likely program invariants — the Daikon system [7]. With KABU, we profile executions of the system to detect which MPs might apply to which methods, and then present these properties for testers to either confirm or reject. We evaluated KABU on two applications, comparing its performance in detecting MPs to that of 23 students from the University of Pennsylvania trained in the task. In our preliminary evaluation, KABU can infer MPs in the methods that are difficult for students to identify.

To further study the efficacy of KABU in detecting MPs, we used it to aid in regression testing. Developers perform regression testing as they modify systems in order to minimize the chances of accidentally introducing a new bug to existing code. Several techniques have been proposed for automating the regression testing process. The work of [1] compared the program invariants detected by Daikon between multiple versions of software. The approach of [16] applied the metamorphic relations from a correct version of software to detect mutants in the following versions.

We built on this approach for automating regression testing by applying KABU to successive versions of the same software and detecting possible MPs in each version, a technique that we call Metamorphic Differential Testing (MDT). KABU flags *Regressed Properties*, which are MPs only owned by the n_{th} version of software and *Progressed Properties*, which are MPs only owned by the $n + 1_{th}$ version, for human review. We applied MDT to two classification algorithms in six versions of the popular ML library Weka. The change logs from Weka verified that the MPs inferred by KABU can detect the changes in correctness in the programs without a test oracle.

This paper is organized as follows: Section II presents an overview of metamorphic testing, and Section III presents our approach to automatically detecting likely MPs. "Likely" is in the sense that the inferred properties hold for the sample executions but not for every possible ones. Section IV presents our preliminary experimental evaluation of KABU. We conclude with a discussion of related work in Section V.

II. BACKGROUND

A. Metamorphic Testing

Metamorphic testing [3] was created as a technique for amplifying the original test suites that may not cover some

978-1-4673-7022-6/15 $31.00 © 2015 IEEE
DOI 10.1109/AST.2015.19

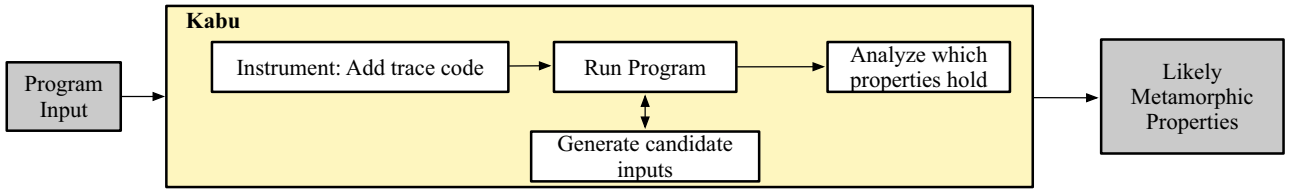

Figure 1: **High Level Overview of KABU:** First, KABU instruments the system by adding tracing code. As the program executes, KABU generates new inputs and feeds these inputs to sandboxed executions of each unit, finally comparing the output states of each of these transformed executions to determine which properties hold.

potential bugs. With metamorphic testing, it is possible to generate additional test cases given a suite of test cases, even if that original test suite does not expose any faults. In metamorphic testing, testers first define metamorphic relations or properties for the system under test. Each metamorphic relation defines a transformation t that can be applied to the input x of one executing program f, so that we can predict the output of $f(t(x))$ given only the output of $f(x)$. From each metamorphic relation, we could generate an infinite number of new test inputs and new test oracles from a single test case. As a simple example, consider a method that computes the average of a set of numbers: one metamorphic relation would state that if we double the values of the numbers given to the method, the output should double as well.

We can apply the relation to an arbitrary execution of the system under test, rather than applying the metamorphic relation to a known test case. While we may not know the correctness of the original execution, we can still identify faults by seeing if the output from the transformed input does not match the expected output given the relation. This enables metamorphic testing to be applied to software for which there is no test oracle — software for which we do not know what the output should be.

B. Regression Testing

As software evolves and is maintained, features are added and bugs are fixed. During this process, developers hope to ensure that new faults are not introduced into old code. Regression testing aims to identify such potential faults via the use of a sufficiently broad test suite for original program P, such that a new version of P, P', can be held to the same test suite. Ideally, such a test suite exists. In practice, it can be very difficult to build a sufficiently high-coverage test suite to detect all possible faults. While automated tools exist for generating unit test suites, they still fall victim to the oracle problem [5]: writing test oracles for each unit may be just as time consuming as writing the test suite.

Differential testing is a promising approach to improving automated regression testing [6]. In general, the goal in differential testing is to compare the output of P and the new version of P, P', in order to detect changes. A more detailed comparison of KABU with other tools for regression testing and for metamorphic testing appears in §V.

III. APPROACH

Before detailing KABU, we define MP that considers both return values and output states of methods. The output state of a method is the set of *Class variables* (CV), *Instance variables* (IV) and *Local variables* (LV): $OS = \{CV, IV, LV\}$. A Metamorphic Property (MP) can be defined as $MP = T + [input'] + C + V$, where T is a input transformer in KABU, $[input']$ is a set of the original input $[input]$ transformed by T, C is a output checker and V is a variable in the OS or the return value that passes the examination of C.

KABU infers MPs by observing method inputs and output states at the variable level during execution, then cross-referencing these values against a pre-defined (but extensible) list of known sorts of MPs. Take a program that has one instance variable, arr, which is an array, for example. One method, $f(x)$, of this program takes an integer as the input, multiplies every element in arr by this integer, sums up arr and returns the sum as the output. If KABU happens to observe two executions of this method, $f(x)$ and $f(2x)$, two possible MPs can be identified: if each input doubles, the return value and every element in arr are expected to double as well.

It may seem contrived to imagine that during the profiling of the system under test, such inputs would occur. To increase our likelihood of identifying likely MPs, KABU systematically injects additional inputs that could expose such properties. For instance, in this case, KABU would observe the execution of $f(x)$, and would then automatically execute (in a sandboxed environment) $f(2x)$ to observe the output states. In this way, we imagine that the profiling inputs for KABU could vary widely, from existing unit tests to randomly generated inputs.

As shown visually in Figure 1, the approach of KABU inferring likely MPs consists of four main steps: 1) Instrumentation, 2) Input Transformation, 3) Program Execution, and 4) Output State Comparison.

Step 1, Instrumentation, occurs before execution to insert stub code in the system under test to support tracing and input transformation. In Step 2, we generate additional inputs to the system in order to expose MPs that might not otherwise be exposed. Next, we execute the system with the original inputs and all transformed inputs. Executions with transformed inputs occur in sandbox containers, allowing them to contain all output states from those executions so as not to affect each other. Output State Comparison occurs after method execution, and gathers the output states of the execution for each test unit to identify which MPs were observed at the variable level.

We have implemented KABU in Java using the ASM [2] bytecode manipulation library. The remainder of this section describes in detail our approach to constructing KABU.

A. Instrumentation

KABU instruments the system under test to support tracing method calls, including all inputs and returns. KABU can be

used in a focused manner, only identifying properties of target methods as directed by developers. At the entry point to each method for which it is to detect MPs, KABU inserts a callback to a runtime interceptor to record the execution of the method along with all of its arguments.

In addition to detecting MPs between method arguments, KABU also generates properties by observing the output state (e.g., static or instance fields) of a method. KABU records fields and variables in target methods and their owner classes or objects to support such an analysis. This approach allows us to consider only the output state that is relevant to the methods under scrutiny, constraining our possible search space for MPs.

B. Input Transformation

We do not expect that the inputs provided to the application during profiling will be sufficient to detect MPs. Therefore, KABU amplifies these inputs by applying various transformations to them. KABU's transformations are based on the transformations that seem to occur frequently in metamorphic relations as identified in prior work [12].

All of our input transformers operate on relatively primitive data types (numbers, Strings, Collections, and arrays). To support detecting MPs on methods that take complex arguments, KABU allows developers to provide a simple adaptor to map from a complex input to a type that KABU can understand. We built several adaptors to complete our experiments.

We have built five input transformers for use in our experiments, described below. The input transformers are completely pluggable and simple to write — on average, each input transformer had less than 100 lines of code.

- **Multiplier:** This transformer multiplies each element in the input by a constant value. If the input data is a String, the Multiplier repeats the same String by a constant value.
- **Adder:** This transformer adds a constant value to each element in the input. If the input data is a String, the Adder treats the constant value as an ASCII code and appends it to the String.
- **Negator:** This transformer is a special case of Multiplier: it multiplies each element in the input by -1 (and applies only to numeric inputs).
- **Shuffler:** This transformer changes the order of elements in the input. This transformer applies only to arrays, Java Collections and Strings. If the input is of String type, the input will be converted to a character array to transform.
- **Reverser:** This transformer inverts the order of elements in the input. Similar to Shuffler, this transformer functions only on arrays, Collections and Strings (which are first converted to character arrays, then reversed).

KABU applies every transformer to each combination of inputs of the method at runtime. Each additional execution occurs in a sandboxed environment to prevent side effects.

C. Program Execution and Profiling

During profiling execution, KABU dynamically monitors the method under scrutiny, via the hooks inserted during the Instrumentation phase. Figure 2 shows an overview of KABU's actions during the execution of a single method. First, KABU forks execution into two threads: the original thread, which executes the method call with the original input, and a child thread to supervise transforming the input and executing the method with this new input. KABU applies each transformer to each combination of inputs and executes the original method using each new input in a separate sandbox, ensuring that any side effects from such executions do not affect the execution of the main program under test. The result of each execution is logged in memory for the next step, *Output State Comparison*.

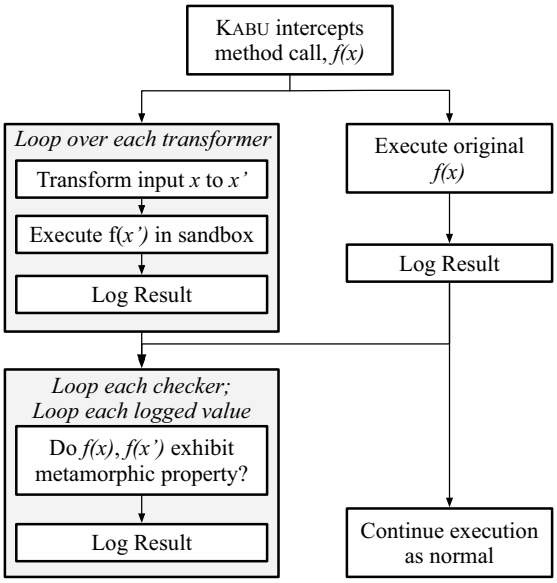

Figure 2: **Detecting properties for individual methods:** Upon invocation of a method call, KABU intercepts the call, forking execution so that while the method f is executed with the original input x, x is transformed using various pluggable transformers. For each transformed x', KABU automatically executes $f(x')$ in a lightweight sandbox. After all results are logged, each pluggable checker module compares the return values and output states of each $f(x')$ with $f(x)$ in order to identify which metamorphic properties may hold.

D. Output State Comparison

The final phase of KABU's process for detecting likely MPs is to analyze the output states of each execution of the same method with different inputs. This analysis occurs at the end of execution after all possible inputs have been seen. In addition to checking the output (return value) of a method, this process also compares relevant output states. For CV and IV, KABU checks the class and the object that the method belongs to. For LV, KABU checks the local variables defined in the method, which are recorded by the variable recorder within KABU.

KABU provides six state comparing checkers similar to its input transformers. Just as the input transformers rely on adapters to support complex data types, so do the checkers. Each pair of inputs is processed through each checker, allowing for all possible combinations. The $f(x)$ used in defining checkers can be the return value or a variable in OS.

- **Identity checker:** This checker determines if $f(x) = f(x')$, and functions for arrays, Collections, Strings and scalar elements.

- **Multiplicative checker:** This checker determines if there is a linear relationship between $f(x)$ and $f(x')$, by identifying if there is a common value d such that for every element in $f(x)$ and $f(x')$, $f(x) * d = f(x')$.
- **Negative checker:** This checker is a special version of multiplicative checker. It checks if $f(x) * -1 = f(x')$.
- **Additive checker:** This checker determines if there is a constant offset between $f(x)$ and $f(x')$, c such that for every element in $f(x)$ and $f(x')$, $f(x) + c = f(x')$.
- **Shuffling checker:** This checker determines if $f(x)$ and $f(x')$ are both arrays, Collections or Strings which contain the exact same elements, but may be out of order.
- **Reversible checker:** This checker determines if $f(x)$ is equivalent to the reversal of $f(x')$.

We constructed the checkers based on our previous experiences in detecting MPs, but they are un-decidable to be a complete set of all relationships. However, we evaluated the efficacy of combining the checkers with the transformers and found the preliminary results to be promising.

IV. Evaluation

To demonstrate the capability of KABU to extract MPs dynamically, we designed two experiments. The first experiment compares the MPs inferred by KABU with those identified by a group of 23 students trained at the task. In the second experiment, we detect changes to MPs between several versions of the same method. Then we compare our results to notes in the application's version control change logs to identify if these property differences truly detect changes or bugs in these machine learning methods without a test oracle.

A. Discovering Metamorphic Properties

We used two simple algorithms: Knapsack and Superstring as evaluation subjects for MP extraction. The Knapsack application takes a list of items with values and weights, and the capacity constraint on weights, as the input. The combination of items with the maximum values without exceeding the weight constraint is returned as the output. The Superstring application returns the shortest common string as the output given a list of strings as the input.

We evaluate the identification rate (IR), which we define as the total number of properties detected by a technique divided by the number of the known properties, and show our results in Table I. The ground-truth for MPs was created by the authors. Compared with the students (32%), KABU infers most MPs (94%) of the known properties. Most of the properties found by KABU but not by the students were state-related. KABU detected only one and zero false positives for the Knapsack and Superstring applications, respectively.

Table I: **The comparison of Identification Rate (IR) between student subjects and KABU**. For each experiment, we show also the number of properties detected in parentheses.

Application	Known Properties	IR(Students)	IR(KABU)
Knapsack	31	0.32(10)	0.94(29)
Superstring	16	0.31(5)	0.94(15)

Table II: **Evaluation of Metamorphic Differential Testing on Weka.** In all cases where a Metamorphic Property changed, the change was tied to a functional change in the code.

(a) LogitBoost

Version Pair	Prop. Diff.	Change Log
3.6.5 vs 3.6.6	0	
3.6.6 vs 3.6.7	0	
3.6.7 vs 3.6.8	0	
3.6.8 vs 3.6.9	12	*Changed resampleWithWeights() to use Walker's alias method. The old implementation did not implement sampling with weights correctly.*
3.6.9 vs 3.6.10	0	

(b) Decorate

Version Pair	Prop. Diff.	Change Log
3.6.5 vs 3.6.6	0	
3.6.6 vs 3.6.7	8	*Restored Prem's original defaults (from the his [sic] paper) for number of iterations and desired ensemble size.*
3.6.7 vs 3.6.8	0	
3.6.8 vs 3.6.9	0	
3.6.9 vs 3.6.10	0	

B. Metamorphic Differential Testing

One limitation of our approach is that the properties that we propose as likely MPs may not necessarily be valid — they require developers' supervision to determine if they are true properties (a limitation shared by previous approaches to automatically detect likely program invariants [7]). To support a more automated use of KABU, we propose *Metamorphic Differential Testing* (MDT), a form of automated regression testing. Our observation is that if a property Pr is only observed in one of the two consecutive versions (n_{th} and $n+1_{th}$) of a method M, then there was a functional change to M that affected this property. The MPs that are observed only in the n_{th} or $n+1_{th}$ version are named *Regressed Property* and *Progressed Property*, respectively. In this case, we believe that it's irrelevant whether the observed property Pr was truly a MP — in either case, it still is a way of encapsulating program behavior. KABU infers the MPs for each provided version of the method under test, and outputs all regressed and progressed properties between two consecutive versions.

The concept of MDT is similar to [16], which applied the metamorphic relations from a correct version of software to execute fault analysis on the next version. It is hard for us to identify which version of software is bug free. The $n + 1_{th}$ version might either fix a bug from the n_{th} version or introduce a new bug, so MDT only reports the regressed and progressed properties between versions for developers to review. Because KABU infers the MPs at the variable level, MDT can flag potential bugs that impact specific variables but not necessarily the return value of a method.

To demonstrate the efficacy of MDT, we applied KABU to six versions of two classification algorithms from the Weka toolkit library: LogitBoost and Decorate. Then we observed the differences in MPs between versions and checked the change logs to verify if a change or a bug in the method had been fixed. To execute and monitor the core methods,

`buildClassifier`, of these two classification algorithms, we used the *iris* dataset provided by Weka.

The result of applying MDT on Weka is in Table II. For the LogitBoost algorithm, the fix of the method `resampleWithWeights` in version 3.6.9 caused 12 property differences between 3.6.8 and 3.6.9. However, there was no property difference before 3.6.8 and after 3.6.9. For the Decorate algorithm, the change of the iteration setting in 3.6.7 resulted in eight property differences between 3.6.6 and 3.6.7. There was no property difference detected by KABU before 3.6.6 and after 3.6.7.

The preliminary result of this experiment supports that KABU can detect changes or potential bugs in versions of applications without a test oracle. The problem of LogitBoost mentioned in Table II was introduced in Weka since 3.1.7, but it had not been fixed until 3.6.9 after several years. One reason may be the lack of a test oracle so that it's hard for testers to find this bug. Even if we applied KABU alone to this algorithm, testers would still have to manually verify the validity of the MPs. However, MDT reports only the differences of the MPs, so testers can focus on checking if such differences are caused by bugs without verifying these MPs.

C. Discussion and Future Work

Our preliminary results indicate that KABU may be an effective tool for detecting MPs. However, there is still much work to do to continue to enhance it. While we have not formally measured the runtime overhead of KABU in detecting MPs, we have observed it to be less than humans take in identifying the same. As next steps of KABU, we plan to amplify the input transformation by incorporating the *Input State* (not only input parameters) of method, as we already checked the *Output State*, and conduct large-scale experiments to evaluate the inference capability of KABU on MPs.

V. Related Work

In most previous research [3], [4], [13], [14], the MPs were selected by hand. The approach proposed by [16] can automatically infer the polynomial metamorphic relations between the input and output of a method. Instead of observing solely the output of a method, KABU considers the overall output state of a method and infers MPs at the variable level that may include static, instance and local variables. The system devised by [9] used the features of a method's control flow graph as the dataset and applied classification algorithms to predict if the pre-defined MPs hold in a method. This approach requires the prior knowledge from humans for training the classifier. KABU infers MPs directly in the search space defined in §III without requiring prior knowledge.

Differential unit testing [6] is similar to KABU in that it can identify functional differences between several versions of code, but compares raw outputs between versions, rather than comparing the MPs that hold over various versions. The Diffut framework [15] similarly supports comparing unit changes by executing several versions of the same unit. We believe that because KABU compares changes in MPs (and not actual outputs) between versions, it will be less sensitive to minor changes between versions than both of these approaches, potentially yielding fewer false positives. The concept of MDT is also similar to [16] as discussed in §IV-B.

VI. Conclusion

In this paper, we proposed a novel approach, KABU, which can dynamically infer Metamorphic Properties (MPs) that include both return values and end states of methods in programs. These MPs are helpful for developers to test programs without a test oracle such as machine learning applications. A testing concept, *Metamorphic Differential Testing* (MDT), is built upon KABU. MDT compares the MPs between different versions of the same application, and reports the differences that may be bugs. The preliminary results of our experiments showed that KABU can infer MPs that are difficult for students to identify. With these MPs, KABU/MDT detected the changes and bugs in two classification algorithms reported in the logs of the Weka library.

VII. Acknowledgments

Su, Bell and Kaiser are members of The Programming Systems Laboratory, which is funded in part by NSF CCF-1302269, CCF-1161079, and NIH U54 CA121852. The authors thank Emily Boggs for conducting the user study.

References

[1] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. ISSTA '06, pages 169–180. ACM, 2006.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[3] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Dept. of Computer Science, HKUST, 1998.

[4] T. Y. Chen, C.-a. Sun, G. Wang, B. Mu, H. Liu, and Z. Wang. A metamorphic relation-based approach to testing web services without oracles. *Int. J. Web Serv. Res.*, 9(1):51–73, Jan. 2012.

[5] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, ACM, pages 254–257, New York, NY, USA, 1981. ACM.

[6] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. FSE-14. ACM, 2006.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[8] D. Hoffman. A Taxonomy for Test Oracles. In *11th International Software Quality Week*, May 1998.

[9] U. Kanewala and J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. ISSRE '13, 2013.

[10] H. Liu, F. Kuo, D. Towey, and T. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE TSE*, 2013.

[11] C. Murphy, J. Bell, F.-H. Su, and G. Kaiser. Metamorphic runtime checking of applications without test oracles. Technical Report CUCS-023-13, Dept. of Computer Science, Columbia University, 2013.

[12] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. SEKE '08, 2008.

[13] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil. On effective testing of health care simulation software. SEHC '11, 2011.

[14] C. Murphy, K. Shen, and G. Kaiser. Automated system testing of programs without test oracles. ISSTA '09, 2009.

[15] T. Xie, K. Taneja, S. Kale, and D. Marinov. Towards a framework for differential unit testing of object-oriented programs. AST '07, 2007.

[16] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. ASE '14. ACM, 2014.