HKU CS Tech Report TR-2009-10

Postprint of acticle in *IEEE Transactions on Software Engineering* **37** (1): 109–125 (2011)

Semi-Proving: An Integrated Method for Program Proving, Testing, and Debugging

Tsong Yueh Chen, *Member*, *IEEE*, T.H. Tse, *Senior Member*, *IEEE*, and Zhi Quan Zhou

Abstract—We present an integrated method for program proving, testing, and debugging. Using the concept of metamorphic relations, we select necessary properties for target programs. For programs where global symbolic evaluation can be conducted and the constraint expressions involved can be solved, we can either prove that these necessary conditions for program correctness are satisfied or identify all inputs that violate the conditions. For other programs, our method can be converted into a symbolic-testing approach. Our method extrapolates from the correctness of a program for tested inputs to the correctness of the program for related untested inputs. The method supports automatic debugging through the identification of constraint expressions that reveal failures.

Index Terms—Software/program verification, symbolic execution, testing and debugging.

1 Introduction

The correctness of software has always been a major concern of both researchers and practitioners. According to Hailpern and Santhanam [37], the cost of proving, testing, and debugging activities "can easily range from 50 to 75 percent of the total development cost." Program proving suffers from the complexity of the proofs and problems in automation even for relatively simple programs. Program testing therefore remains the most popular means of verifying program correctness [6].

A fundamental limitation of program testing is the *oracle problem* [55]. 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" [5]. Unfortunately, an ideal oracle may not necessarily be available or may be too difficult to apply. For example, for programs handling complex numerical problems, such as those solving partial differential equations, people may not be able to decide whether the computed results of given inputs are correct [15]. In cryptographic systems, very large integers are involved in the public key algorithms. Practically, it is too expensive to verify the computed results [53]. When testing a Web search engine, it is practically impossible to decide whether the returned results are complete [60]. In object-oriented software testing, it is difficult to judge whether two objects are observationally equivalent [12], [13].

The inability to obtain an ideal oracle, however, does not mean that the relevant program cannot be tested. This is because testers can often identify some necessary properties of the program and verify the results of test case executions against these properties. According to the survey by Baresi and Young [5] and the practice of the software testing community, necessary properties (including assertions embedded in a program under test) may also be considered to be a type of oracle. For instance, when testing numerical programs, a frequently employed approach is to check whether such programs satisfy certain expected identity relations derived from theory. Take the program computing e^x as an example. A property that can be employed in testing is $e^x \times e^{-x} = 1$. This kind of identity relation was extensively used to test numerical programs [20].

The techniques of *program checker* [7] and *self-testing/correcting* [8], [47] also make intensive use of expected identity relations of the target functions to test programs and check outputs automatically and probabilistically. To construct a self-tester/corrector, for instance, the fundamental technique is to exploit the properties that uniquely define the target function and to test that the program satisfies these properties for random inputs. Basically, two properties are checked, namely, linear consistency and neighbor consistency [8], which are identity relations among multiple executions of the program.

This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project no. 717308), a Discovery Grant of the Australian Research Council (project no. DP 0771733), and a Small Grant of the University of Wollongong.

T.Y. Chen is with the Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn, Victoria 3122, Australia. E-mail: tychen@swin.edu.au.

T.H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. E-mail: thtse@cs.hku.hk.

All correspondence should be addressed to Dr. Zhi Quan Zhou, School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW 2522, Australia. E-mail: zhiquan@uow.edu.au.

In the area of fault tolerance, there is a closely related technique called *data diversity* [3], which is based on the novel idea of running the same program on reexpressed 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 it has only been advocated as a fault tolerance technique. Consequently, properties used in data diversity are intrinsically limited to identity relations.

We note that the above methods do not address debugging issues with regard to locating and correcting faults in program code. In this paper, we present an integrated method that covers proving, testing, and debugging. While our approach can handle debugging on its own ground, it can also be applied together with the techniques in related work outlined above, so that 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.

The idea of checking the expected properties of target systems without being restricted to identity relations has been employed in *metamorphic testing* [14] and the testing of observational equivalence and nonequivalence of objects [13], [54]. In particular, we will explain metamorphic testing here because some of its concepts will be adopted in this paper. Suppose we run a set of test cases that have been generated according to some test case selection strategies. 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*. They can be any expected relations (including but not limited to identities) among the inputs and outputs of more than one execution of the same program, rather than the relations between the input and output in a *single* execution. In a program $P(a, b)$ computing the integral of a function f from a to b, for example, the property $\int_a^c f(x)\,dx + \int_c^b f(x)\,dx = \int_a^b f(x)\,dx$ can be identified as a metamorphic relation. The testing will involve three executions, namely, $P(a, c)$, $P(c, b)$, and $P(a, b)$. Even if the correctness of an individual output is unknown, the metamorphic relation can still be checked. If the computed results of test cases do not satisfy the expected relation after rounding errors have been taken into account, it will indicate some defects in the program. We should note that metamorphic relations are only necessary conditions and hence may not be sufficient for program correctness. In any case, this is a limitation of all program-testing methods and will be further discussed in this paper. Follow-up studies on metamorphic testing have been conducted in [33], [34]. The idea of metamorphic testing has also been applied to fault-based testing [18] and the testing of pervasive computing [10], Web services [11], and search engines [60].

Like other testing methods, metamorphic testing is limited by the inability of obtaining a *reliable test set* [40]. Even when a program satisfies a given metamorphic relation for all conceivable test cases, it remains unknown whether the program satisfies this relation throughout the input domain. Furthermore, if the given metamorphic relation can indeed be proven for the program, can this result be exploited further to establish an even higher confidence on the correctness of the program or be combined with conventional testing to improve test efficiency? On the other hand, if a program does not satisfy a necessary property for some test cases, how can we debug it automatically?

We address these problems by means of a *semi-proving* method, which integrates program proving, testing, and debugging.[1] This paper presents the development, implementation, and empirical evaluation of the method.

Using the concept of metamorphic relations, we select necessary properties for the target program. Then, we perform *symbolic executions* [19], [21], [24], [45]. This is because the output of a symbolic execution is more informative than that of executing a concrete input, as a symbolic input represents more than one element of the input domain. In recent years, symbolic execution techniques have been studied intensively not only in software engineering, but also in the areas of programming languages and high-performance compilers [9], [25], [26], [27], [29], [36]. Sophisticated techniques and automated tools for symbolic-analysis and path-constraint simplifications have been developed to facilitate more effective parallelism and optimization of programs. They can support data structures ranging from arrays to dynamically allocated complex data types [26]. Commercial computer algebra systems available in the market, such as MATHEMATICA [56], are being strengthened with more and more powerful symbolic-computing abilities. In the rest of this paper, we will use the term *symbolic executions* to refer to the executions of selected paths with selected symbolic inputs, and the term *global symbolic evaluation* for the executions of *all* possible paths of the program with symbolic inputs covering the entire input domain [19].

For programs where global symbolic evaluation can be performed and the constraint expressions involved can be solved, we can either prove that these necessary conditions for program correctness are satisfied, or identify *all* inputs that cause the violation of the conditions. The failure-causing inputs will be represented by constraint expressions instead of individual failure-causing inputs. For programs that are too complex for global symbol evaluation or constraint solvers, our approach can still be applied as a symbolic-testing approach. The technique can also be combined with concrete test cases to extrapolate the correctness to related untested inputs. For all programs, the diagnostic information generated for the detected failure supports automatic debugging.

In Section 2, we will present the semi-proving method and its features. In Section 3, we will present the empirical evaluation results using the *replace* program, which is the largest and most complex among the Siemens suite of programs [41]. Also included will be a brief description of

[1] A preliminary version of semi-proving was proposed in ISSTA 2002 [17].

the prototype implementation of our method. Section 4 will discuss the limitations of our method and the treatments to alleviate some of its problems. Section 5 compares our method with related work. Section 6 will conclude the paper.

2 Our Methodology

2.1 Overview

Our method employs symbolic-evaluation and constraint solving techniques [19], [22], [23], [26], [27], [29], [42], [44]. The procedure for semi-proving is outlined as follows: Given a program P under test and a metamorphic relation (MR), we first take a symbolic input vector I and conduct a global symbolic evaluation of P on I. Let $O_1, O_2, \ldots, O_n$ be the symbolic outputs and let C_i be the path condition of O_i for $i = 1, 2, \ldots, n$. For ease of presentation, we assume that the given MR involves only two executions of P. According to the MR, we generate a second symbolic input vector I', on which a second global symbolic evaluation will be conducted. Suppose the symbolic outputs are $O'_1, O'_2, \ldots, O'_m$ under path conditions $C'_1, C'_2, \ldots, C'_m$, respectively. (Note that m is not necessarily equal to n. For instance, if I represents a single integer, and I' represents the absolute value of I, any path condition related to negative input numbers will not appear in $C'_1, C'_2, \ldots, C'_m$.) Then, for each C_i $(i = 1, 2, \ldots, n)$ and C'_j $(j = 1, 2, \ldots, m)$, we evaluate the conjunction of C_i and C'_j. If it is not a contradiction, then we check whether the MR is satisfied under this conjunction. If a violation is detected, it will be reported and further debugging information will be generated.

We would like to illustrate various features of the semi-proving methodology in the following sections. Section 2.2 will demonstrate how to apply semi-proving to prove that the program satisfies the metamorphic relation for the entire input domain. In Section 2.3, we will illustrate how our method can be used to extrapolate from the correctness of a program for tested *symbolic* inputs to the correctness of the program for related symbolic inputs that have not been tested. In Section 2.4, we will show how our method can automatically detect program defects by identifying *all* inputs for which the program fails to satisfy the metamorphic relation. In Section 2.5, we will explain how semi-proving supports automatic debugging. In Section 2.6, we will discuss treatments for loops. In Section 2.7, we will further discuss how to extrapolate from the correctness of a program for tested *concrete* inputs to its correctness for related concrete inputs that have not been tested.

2.2 Proving Metamorphic Relations

Semi-proving proposes the proving of selected necessary properties for program correctness, expressed as metamorphic relations of the specified function. Fig. 1 shows a program *Med*. It accepts three integers as input and returns the median as output. The program is adapted from [50], where it was used as a worst-case example to illustrate the technique of constraint-based test case generation for mutation testing. Let us use this example program to illustrate how semi-proving proves metamorphic relations.

TABLE 1
Results of Global Symbolic Evaluation of *Med* (a, b, c)

Path	Path Condition	Result
P_1: (1,2,3,4,11)	C_1: $a < b < c$	b
P_2: (1,2,3,5,6,11)	C_2: $b \leq a < c$	a
P_3: (1,2,3,5,11)	C_3: $b < c \leq a$	c
P_4: (1,2,7,8,11)	C_4: $c \leq b < a$	b
P_5: (1,2,7,9,10,11)	C_5: $c < a \leq b$	a
P_6: (1,2,7,9,11)	C_6: $a \leq c \leq b$	c

TABLE 2
Results of Global Symbolic Evaluation of *Med* (a, c, b)

Path	Path Condition	Result
P'_1: (1,2,3,4,11)	C'_1: $a < c < b$	c
P'_2: (1,2,3,5,6,11)	C'_2: $c \leq a < b$	a
P'_3: (1,2,3,5,11)	C'_3: $c < b \leq a$	b
P'_4: (1,2,7,8,11)	C'_4: $b \leq c < a$	c
P'_5: (1,2,7,9,10,11)	C'_5: $b < a \leq c$	a
P'_6: (1,2,7,9,11)	C'_6: $a \leq b \leq c$	b

Let *median* denote the specification function for the program *Med*. An obvious metamorphic relation is *median* $(I) =$ *median* $(\pi(I))$, where I is the tuple of parameters, and $\pi(I)$ is any permutation of I. For example, *median* $(1, 2, 3) =$ *median* $(3, 1, 2)$. We would like to prove that the program *Med* preserves this property for *all* elements in the input domain.

We need some knowledge of the problem domain to facilitate the proving process. From group theory [38], we know that all of the permutations of I, together with the compositions of permutations, form a *permutation group*. It is also known that $\tau_1(a, b, c) = (a, c, b)$ and $\tau_2(a, b, c) = (b, a, c)$ are *generators* of the permutation group. In other words, any permutation of I can be achieved by compositions of the transpositions τ_1 and τ_2. For instance, the permutation (b, c, a) can be achieved by applying τ_2 followed by τ_1 to the tuple (a, b, c). Hence, in order to prove that *Med* $(I) =$ *Med* $(\pi(I))$ for any input tuple I and any permutation π of I, we need only prove two identities, namely, *Med* $(a, b, c) =$ *Med* (a, c, b) and *Med* $(a, b, c) =$ *Med* (b, a, c).

It is straightforward to conduct global symbolic evaluation [19] on the program *Med*. Let the symbolic input be (a, b, c), which represents any triple of integers. The results are shown in Table 1. There are, altogether, six possible execution paths. For any valid input, one and only one of the six path conditions C_1 to C_6 will hold.

We recognize that we can directly prove or disprove this simple program using the results of Table 1 since there is an oracle for all the symbolic outputs. Nevertheless, we would like to use the program to illustrate how to perform semi-proving without assuming the knowledge about the correctness of each individual output.

Let us apply the transformation τ_1 to the *initial* symbolic input (a, b, c) to obtain a *follow-up* symbolic input (a, c, b). We will refer to the executions on the initial and follow-

```
     int Med(int u, int v, int w) {
        int med;
 1      med = w;
 2      if (v < w)
 3         if (u < v)
 4            med = v;
           else {
 5            if (u < w)
 6               med = u; }
        else
 7         if (u > v)
 8            med = v;
           else {
 9            if (u > w)
10               med = u; }
11      return med; }
```

Fig. 1. Program *Med*

```
     int M̅e̅d̅(int u, int v, int w) {
        int med;
 1      med = w;
 2      if (v < w)
 3         if (u < v)
 4            med = v;
           else {
 5            if (u < w)
 6               med = u; }
        else
 7         if (u > v)
 8            med = v;



11      return med; }
```

Fig. 2. Program $\overline{\mathit{Med}}$

up inputs as the *initial execution* and the *follow-up execution*, respectively. The results of the follow-up global symbolic evaluation of *Med* (a, c, b) are listed in Table 2.

Our target is to prove that *Med* $(a, b, c) =$ *Med* (a, c, b) for any input tuple (a, b, c). As indicated by Table 1, we need to prove the metamorphic relation for six cases according to the path conditions C_1 to C_6. Consider condition C_1. From Table 1, when condition C_1 holds, the output of *Med* (a, b, c) is the variable b. Table 2 shows that the output of *Med* (a, c, b) is also b for paths P'_3 and P'_6, but not b for paths P'_1, P'_2, P'_4, and P'_5. Thus, we need to know whether P'_1, P'_2, P'_4 and P'_5 will be executed when C_1 holds. In other words, we need to check each of the combined conditions "C_1 **and** C'_1," "C_1 **and** C'_2," "C_1 **and** C'_4," and "C_1 **and** C'_5." We find that each of them is a contradiction. Thus, *Med* (a, c, b) also returns the variable b when condition C_1 holds.

Referring to Table 1 again, when condition C_2 holds, the output of *Med* (a, b, c) is the variable a. Table 2 shows that the output of *Med* (a, c, b) is also a for paths P'_2 and P'_5, but not for paths P'_1, P'_3, P'_4, and P'_6. We find that each of the combined conditions "C_2 **and** C'_1," "C_2 **and** C'_3," and "C_2 **and** C'_4" is a contradiction. Furthermore, when we evaluate "C_2 **and** C'_6," we obtain "$a = b < c$." In other words, although *Med* (a, b, c) returns the variable a and *Med* (a, c, b) returns the variable b, both of them have the same value. Hence, the metamorphic relation holds when condition C_2 is satisfied.

We can similarly prove that the results of *Med* (a, b, c) and *Med* (a, c, b) are consistent with each other under the conditions C_3, C_4, C_5, and C_6. Thus, we have proven that *Med* $(a, b, c) =$ *Med* (a, c, b) for any input tuple (a, b, c). In the same way, we can prove the metamorphic relation for transposition τ_2, namely, that *Med* $(a, b, c) =$ *Med* (b, a, c) for any input tuple. According to group theory, therefore, *Med* $(I) =$ *Med* $(\pi(I))$ for any input tuple I and any permutation $\pi(I)$.

2.3 Extrapolation of Correctness for Symbolic Test Cases

In the previous section, we illustrated how to prove that the program is correct with respect to a selected metamorphic relation. In situations where the correctness of some outputs can be decided, such as special value cases, we can extrapolate from the correctness for tested inputs to the correctness for related *untested* inputs. Let us, for instance, test the program *Med* with a specific *symbolic* test case (x, y, z) such that $x \leq z \leq y$. The output is z. We can, of course, easily verify that z is a correct output. Hence, the program passes this specific test. Having proven the metamorphic relation in Section 2.2 and tested the correctness of the program for one specific symbolic input, we can extrapolate that the outputs of the program *Med* are correct for all other inputs as follows: Suppose $I = (a, b, c)$ is any triple of integers. Let $\pi(I) = (a', b', c')$ be a permutation of I such that $a' \leq c' \leq b'$. According to the result of the symbolic testing above, *Med* $(a', b', c') =$ *median* (a', b', c'). The fact that (a', b', c') is simply a permutation of (a, b, c) implies that *median* $(a', b', c') =$ *median* (a, b, c). Hence, *Med* (a', b', c') $=$ *median* (a, b, c). On the other hand, it has been proven that *Med* $(a', b', c') =$ *Med* (a, b, c). Therefore, *Med* (a, b, c) $=$ *median* (a, b, c). In this way, we have proven that the outputs of *Med* (a, b, c) are correct for *any* input. In other words, the correctness is extrapolated from tested symbolic inputs to untested symbolic inputs.

When the test case is concrete rather than symbolic, our approach can also be applied to extrapolate the correctness of the program to related concrete inputs that have not been tested. Details of semi-proving for concrete inputs will be discussed in Section 2.7.

2.4 Identifying Program Faults through Metamorphic Failure-Causing Conditions

In this section, we will show how a fault can be detected by semi-proving. Suppose we remove statements 9 and 10 from the program *Med* to seed a *missing path error*[2], which is generally considered as "the most difficult type of error to detect by automated means" [50]. Let us denote the faulty program by $\overline{\mathit{Med}}$, as shown in Fig. 2. The global symbolic evaluation results for the initial symbolic input (a, b, c) and the follow-up symbolic input (a, c, b) are shown in Tables 3 and 4, respectively.

[2] We appreciate that we should use the word "fault" instead of "error" when discussing a missing path in a program. We will, however, continue to use "missing path error" as a courtesy to the pioneers who coined this phrase well before the *IEEE Standard Glossary of Software Engineering Terminology* was published in 1990.

TABLE 3
Results of Global Symbolic Evaluation of $\overline{\mathit{Med}}(a, b, c)$

Path	Path Condition	Result
P_1: (1,2,3,4,11)	D_1: $a < b < c$	b
P_2: (1,2,3,5,6,11)	D_2: $b \leq a < c$	a
P_3: (1,2,3,5,11)	D_3: $b < c \leq a$	c
P_4: (1,2,7,8,11)	D_4: $c \leq b < a$	b
P_5: (1,2,7,11)	D_5: $c \leq b$ **and** $a \leq b$	c

TABLE 4
Results of Global Symbolic Evaluation of $\overline{\mathit{Med}}(a, c, b)$

Path	Path Condition	Result
P'_1: (1,2,3,4,11)	D'_1: $a < c < b$	c
P'_2: (1,2,3,5,6,11)	D'_2: $c \leq a < b$	a
P'_3: (1,2,3,5,11)	D'_3: $c < b \leq a$	b
P'_4: (1,2,7,8,11)	D'_4: $b \leq c < a$	c
P'_5: (1,2,7,11)	D'_5: $b \leq c$ **and** $a \leq c$	b

A failure will be detected when condition D_2 is being checked. When D_2 holds, the following expressions need to be evaluated: (D_2 **and** D'_1), (D_2 **and** D'_3), (D_2 **and** D'_4), and (D_2 **and** D'_5). While the first three are shown to be contradictions, the fourth conjunction is satisfiable. It can be simplified to "$b < a < c$ **or** $b = a < c$." Obviously, when $b = a < c$, $\overline{\mathit{Med}}(a, b, c) = \overline{\mathit{Med}}(a, c, b)$. When $b < a < c$, however, $\overline{\mathit{Med}}(a, b, c) \neq \overline{\mathit{Med}}(a, c, b)$ because $a \neq b$. As a result, all input pairs satisfying "$b < a < c$" are found to cause the failure. The expression "$b < a < c$" is therefore called a *Metamorphic Failure-Causing Condition* (*MFCC*). All MFCCs related to a given metamorphic relation can be similarly identified. Note that, in general, MFCCs are dependent on specific metamorphic relations and are related to different faults.

It should be noted that, in conventional software testing (including metamorphic testing), concrete failure-causing inputs are identified but not the interrelationships among them. Semi-proving, on the other hand, supports debugging by providing explicit descriptions of the interrelationships among metamorphic failure-causing inputs via MFCC. Compared with concrete failure-causing inputs, such interrelationships contain more information about the defects. This is because constraint expressions like "$b < a < c$" (instead of concrete inputs like "$a = 0$, $b = -1$, and $c = 2$") represent multiple (and, possibly, infinitely many) concrete failure-causing inputs and reveal their characteristics.

Compared with metamorphic testing, our method has another advantage in addition to its support of debugging: It has a higher fault-detection capability. Generally speaking, the inputs that exercise the failure paths (that is, the pair of paths corresponding to the initial and follow-up executions that reveal a violation of a metamorphic relation) may not necessarily cause a failure. In other words, concrete test cases in metamorphic testing may not detect a failure even if they have executed such paths. On the other hand, semi-proving guarantees the detection of the failure via symbolic inputs that exercise these paths.

2.5 Debugging

In the previous section we have shown how MFCCs can be automatically generated when a failure is detected. In this section, we will illustrate how to use the generated MFCCs to facilitate debugging.

According to Pezzè and Young [51], "Testing is concerned with fault detection, while locating and diagnosing faults fall under the rubric of debugging." As is widely agreed in the industry, "most of the effort in debugging involves locating the defects" [37]. While we fully concur with this observation, we would like to add that, in many situations, a defect or fault may not be precisely localized to particular statements, as a faulty program may be corrected in many different ways. Hence, "locating the defects" should not simply be interpreted as the identification of faulty statements in a program. We have implemented a verification and debugging system with a different focus. Our system generates diagnostic information on the cause-effect chain that leads to a failure.

2.5.1 Debugging Step 1: Selecting a Failure Path for Analysis

For ease of presentation, let us assume that the MR involves two executions, namely, an initial execution and a follow-up execution. Once a violation of the MR is detected, at least one of these two executions must be at fault. To debug, details of both executions will be reported to the user. The user may choose to instruct the debugger to focus on a particular path. Otherwise the debugger will use a heuristic approach as follows: Let the initial execution path be A and the follow-up execution path be B. Our verification system will continue to verify other paths of the program with a view to identifying more violations of the MR. After that, the debugger will compare paths A and B against the paths involving other violations and calculate the frequencies of occurrence for both A and B. Suppose, for instance, that A involves more violations of the MR than B. Then, our debugger will select A as the focus of debugging and proceed to "Debugging Step 2." We recognize that the heuristic approach may not always correctly identify the genuine failure path. Users, however, can always instruct the debugger to move their focus to the other path if they so prefer.

For the example related to $\overline{\mathit{Med}}(a, b, c)$ and $\overline{\mathit{Med}}(a, c, b)$ as discussed in the last section, our debugger finds that the initial execution (P_2) involves a total of two violations of the MR (as indicated in row 29 of the failure report in Fig. 3), and the follow-up execution (P'_5) involves a total of four violations (as indicated in row 30 of the report.) The latter has therefore been selected. This is indeed the genuine failure path.

2.5.2 Debugging Step 2: Comparing Successful and Failed Executions to Find the Cause of Failure

We define a *metamorphic preserving condition* (MPC) as a condition under which a program satisfies a prescribed metamorphic relation. When we identify an MFCC, a corresponding MPC can also be identified at the same time.

In Section 2.4, for instance, while "$b < a < c$" is detected to be an MFCC, another constraint expression "$b = a < c$" can be identified as the corresponding MPC. A violation of an MR occurs because *some input pairs satisfy an MFCC rather than an MPC*. Hence, if we compare these two conditions, the difference can be considered as the *trigger* of the failure. This trigger can supply us with more clues for the discovery of the nature of the defect. In the simple case illustrated by our example, our debugger finds that the difference between the MFCC and the MPC involves the relationship between a and b: It is $b < a$ in MFCC, but is $b = a$ in MPC.

Once the trigger is identified, the debugger will further compare the two execution logs, including execution traces, path conditions, and so on, and then report the differences as a cause-effect chain that leads to the failure. The final failure report for the $\overline{Med}$ example is shown in Fig. 3. It was automatically generated by our debugger without the need to interact with users. For the sake of presentation, we have slightly modified the line numbers quoted by the report so that they are consistent with those in Fig. 2.

Row 1 of Fig. 3 shows the symbolic input for an initial execution (P_2), consisting of `v1` for u, `v2` for v, and `v3` for w. We note that "1st run" printed in rows 1, 11, 25, 26, and 29 refers to the initial execution. Row 2 records execution trace, that is, function $\overline{Med}$ is entered. Row 3 records that a *true* branch is taken at statement 2 (which is "if (v < w)".) Row 4 reports that a new constraint is added into the current path condition. The constraint is interpreted as –`v3` + `v2` ≤ –1, or `v2` < `v3` as all inputs are integers. Rows 5 to 8 are interpreted similarly. Row 9 records that function $\overline{Med}$ returns at statement 11. Row 10 prints the symbolic value `v1` returned by $\overline{Med}$. Row 11 marks the end of the initial execution.

Note that the words "2nd run" printed in rows 12, 24, 27, 28, 30, and 31 refers to the follow-up execution (P'_5). Row 12 says that the symbolic input is `v1` for u, `v3` for v, and `v2` for w. Row 19 records that the symbolic output is `v2`. Row 20 reports that a failure is detected. Row 21 reports that the expected output is `v1`, as this is the output of the initial execution. Rows 22 and 23 report the **trigger** of the failure, which is equivalent to `v2` < `v1` since all inputs are integers. This means that out of all the inputs that exercise both the initial and follow-up execution paths, some violate the MR and others do not. The unique characteristic for any input that violates the MR is `v2` < `v1`. Therefore, testers need to pay attention to this condition as it is not handled properly by the program $\overline{Med}$. In fact, this is exactly the root cause of the failure, that is, overlooking the case when `v2` < `v1`.

The reason why the follow-up execution instead of the initial execution has been highlighted for debugging (as shown in Fig. 3) is that our debugger detected that the follow-up execution path involves four violations of the MR, whereas the initial execution path involves two violations. This is reported in rows 29 and 30. In other words, the statistics are aggregated from all violations of the MR, although some of the violations are not shown in the figure. Our debugger therefore considers the follow-up execution to be more likely to be relevant to the failure, as reported in row 31. More discussions about failure reports will be given in Section 3. To assist in debugging, the debugger also generates an example of a concrete failure-causing input, as shown in rows 25 to 28, where rows 27 and 28 are highlighted as they belong to the follow-up execution. In fact, the follow-up execution is indeed the genuine failure path.

Row 31 indicates that debugging should be focused on the follow-up execution because, as reported in rows 29 and 30, violations of the MR occur more frequently in the follow-up execution path than in the initial execution.

Note that the debugging message printed can further provide a hint on how to correct the faulty statement. It suggests the symbolic value of the expected output. Any discrepancy from the symbolic value of the actual output can be tracked down to individual symbols, which can be mapped to individual program variables.

2.6 Dealing with Loops

Consider an example program *Area* (f, a, b, v) adapted from [19] and shown in Fig. 4. For ease of presentation, we will omit the fifth parameter *errFlag* of the program *Area* in our discussion. It used the trapezoidal rule to compute the approximate area bounded by the curve $f(x)$, the x-axis, and two vertical lines $x = a$ and $x = b$. Any area below the x-axis is treated as negative. According to elementary calculus, *Area* (f, a, b, v) computes the area from the integral $\int_a^b f(x)\,dx$ if $a \le b$, and from $-\int_a^b f(x)\,dx$ if $a > b$. The computation uses v intervals of size $|b-a|/v$.

Let $g(x) = f(x) + c$, where c is a positive constant. From elementary calculus, we know that if $a \le b$, $\int_a^b g(x)\,dx = \int_a^b f(x)\,dx + c \times (b-a)$, whereas if $a > b$, $\int_b^a g(x)\,dx = \int_b^a f(x)\,dx + c \times (a-b)$ giving $-\int_a^b g(x)\,dx = -\int_a^b f(x)\,dx + c \times (a-b)$. Hence, we can identify the following metamorphic relation for verification:

$$\textit{Area}\,(g,\, a,\, b,\, v) = \textit{Area}\,(f,\, a,\, b,\, v) + c \times |b-a| \tag{1}$$

where $v \ge 1$. Note that "$(b-a)$" in the integration formula has been changed to "$|b-a|$" according to the specification of the program. Having identified the metamorphic relation, we apply global symbolic-evaluation and loop-analysis techniques introduced by Clarke and Richardson [19] to the program *Area* and obtain six categories of execution paths for any initial input (f, a, b, v). The paths, path conditions, and corresponding symbolic outputs are shown in Table 5. For path P_3, the notation $(9, 10, 11, 12, 13)^+$ denotes multiple iterations of the subpath enclosed in brackets.

To prove the metamorphic relation (1), we also need to do a follow-up global symbolic evaluation for *Area* (g, a, b, v). Table 6 shows the evaluation results. If we compare Tables 5 and 6, we find that the two global symbolic evaluations produce identical sets of paths and path conditions. The only difference between the results is the symbolic values of the variable *area*. This is because the change of the first parameter from f to g does not affect the nature of the execution paths.

```
[symbolic input of 1st run] v1, v2, v3
[trace] entry to Med()
[trace] true branch taken, line 2
[pc]    -1 * v3  + 1 * v2  <=  -1
[trace] false branch taken, line 3
[pc]    1 * v1  + -1 * v2  >=  0
[trace] true branch taken, line 5
[pc]    -1 * v3  + 1 * v1  <=  -1
[trace] return from Med(), line 11
[symbolic output]   1 * v1 + 0
-------- end of 1st run --------
[symbolic input of 2nd run] v1, v3, v2
[trace] entry to Med()
[trace] false branch taken, line 2
[pc]    1 * v3  + -1 * v2  >=  0
[trace] false branch taken, line 7
[pc]    -1 * v3  + 1 * v1  <=  0
[trace] return from Med(), line 11
[symbolic output]   1 * v2 + 0
[debugging diagnosis] failure is detected for the above output
[debugging diagnosis] expected output is    1 * v1 + 0
[debugging diagnosis] trigger of failure is as follows:
   -1 * v1  + 1 * v2  <=  -1
-------- end of 2nd run --------
[example input of 1st run]  1, 0, 2
[example output of 1st run] 1
[example input of 2nd run]  1, 2, 0
[example output of 2nd run] 0
[frequency] path of 1st run occurs in violations of MR: 2 time(s)
[frequency] path of 2nd run occurs in violations of MR: 4 time(s)
[current debugging focus] 2nd run
```

Fig. 3. Failure Report for Program $\overline{Med}$

```
/* Program Area uses the trapezoidal rule to compute the area of the region
enclosed by the curve f(x), the x-axis, and the vertical lines x = a and x = b. The
computation uses v intervals of size |b – a|/v. The variable "errFlag" will be set
to "true" if v is less than 1, and set to "false" otherwise. */

 float Area(float (*f)(float), float a, float b, int v, bool & errFlag) {
   float area;
   float h; /* interval */
   float x;
   float yOld; /* value of f(x – h) */
   float yNew; /* value of f(x) */

1  if (v < 1)
2     errFlag = true;
   else {
3     errFlag = false;
4     area = 0;
5     if (a != b) {
6        h = (b – a) / v;
7        x = a;
8        yOld = (*f)(x);
9        while ((a > b && x > b) || (a < b && x < b)) {
10          x = x + h;
11          yNew = (*f)(x);
12          area = area + (yOld + yNew) / 2.0;
13          yOld = yNew;
            }
14       area = area * h;
15       if (a > b)
16          area = –area; } }
17    return area; }
```

Fig. 4. Program *Area*

Since there are six path conditions on either side of relation (1), we need to verify the results for the $6 \times 6 = 36$ combinations of conditions. Because the path conditions are mutually exclusive, 30 of the combinations are contradictions. Hence, we need to consider only six combinations. Furthermore, path P_6 is not relevant because its path condition is "$v < 1$" but the selected metamorphic relation applies to $v \geq 1$. As a result, there are altogether five combinations to be verified, namely, $(P_1,\, P_1')$, $(P_2,\, P_2')$, $(P_3,\, P_3')$, $(P_4,\, P_4')$, and $(P_5,\, P_5')$. By $(P_i,\, P_i')$, we mean that the initial execution is on path P_i and the follow-up execution is on path P_i'.

Consider the combination $(P_3,\, P_3')$. Under their path conditions, the following computations are made:

$$
\begin{aligned}
& \mathit{Area}\,(g,\, a,\, b,\, v) - \mathit{Area}\,(f,\, a,\, b,\, v) \\
&= \Big((f(a) + c) \ / \ 2.0 + (f(b) + c) \ / \ 2.0 \\
&\qquad + \sum_{i=1}^{v-1} (f(a - a \times i/v + b \times i/v) + c) \Big) \times (a/v - b/v) \\
&\quad - \Big(f(a) \ / \ 2.0 + f(b) \ / \ 2.0 \\
&\qquad + \sum_{i=1}^{v-1} f(a - a \times i/v + b \times i/v) \Big) \times (a/v - b/v) \\
&= (2c/2 + c \times (v - 1)) \times (a - b) \ / \ v \\
&= c \times (a - b) \ = \ c \times |b - a|.
\end{aligned}
$$

TABLE 5
Results of Global Symbolic Evaluation of *Area* $(f,\ a,\ b,\ v)$

Path	**Path Condition**	***area***
P_1: (1,3,4,5,6,7,8,9,10, 11,12,13,14,15,16,17)	$v = 1$ **and** $a > b$	$[f(a)\ /\ 2.0 + f(b)\ /\ 2.0] \times (a - b)$
P_2: (1,3,4,5,6,7,8,9,10, 11,12,13,14,15,17)	$v = 1$ **and** $a < b$	$[f(a)\ /\ 2.0 + f(b)\ /\ 2.0] \times (b - a)$
P_3: (1,3,4,5,6,7,8,(9,10, 11,12,13)$^+$,14,15,16,17)	$v > 1$ **and** $a > b$ **and** $k_e = v$, where $k_e = \min\{k \in \mathbb{N}\ :\ k > 1$ **and** $-b + k \times (b - a)\ /\ v + a \leq 0\}$	$[f(a)\ /\ 2.0\ +\ f(b)\ /\ 2.0$ $+ \sum_{i=1}^{v-1} f(a - a \times i/v + b \times i/v)]$ $\times (a/v - b/v)$
P_4: (1,3,4,5,6,7,8,(9,10, 11,12,13)$^+$,14,15,17)	$v > 1$ **and** $b > a$ **and** $k_e = v$, where $k_e = \min\{k \in \mathbb{N}\ :\ k > 1$ **and** $-b + k \times (b - a)\ /\ v + a \geq 0\}$	$[f(a)\ /\ 2.0 + f(b)\ /\ 2.0$ $+ \sum_{i=1}^{v-1} f(a - a \times i/v + b \times i/v)]$ $\times (b/v - a/v)$
P_5: (1,3,4,5,17)	$v \geq 1$ **and** $a = b$	0
P_6: (1,2,17)	$v < 1$	undefined

TABLE 6
Results of Global Symbolic Evaluation of *Area* $(g,\ a,\ b,\ v)$

Path	**Path Condition**	***area***
P'_1: (1,3,4,5,6,7,8,9,10, 11,12,13,14,15,16,17)	$v = 1$ **and** $a > b$	$[(f(a) + c)\ /\ 2.0 + (f(b) + c)\ /\ 2.0]$ $\times (a - b)$
P'_2: (1,3,4,5,6,7,8,9,10, 11,12,13,14,15,17)	$v = 1$ **and** $a < b$	$[(f(a) + c)\ /\ 2.0 + (f(b) + c)\ /\ 2.0]$ $\times (b - a)$
P'_3: (1,3,4,5,6,7,8, (9, 10,11,12,13)$^+$,14,15,16, 17)	$v > 1$ **and** $a > b$ **and** $k_e = v$, where $k_e = \min\{k \in \mathbb{N}\ :\ k > 1$ **and** $-b + k \times (b - a)\ /\ v + a \leq 0\}$	$[(f(a) + c)\ /\ 2.0 + (f(b) + c)\ /\ 2.0$ $+ \sum_{i=1}^{v-1} (f(a - a \times i/v + b \times i/v) + c)]$ $\times (a/v - b/v)$
P'_4: (1,3,4,5,6,7,8,(9,10, 11,12,13)$^+$,14,15,17)	$v > 1$ **and** $b > a$ **and** $k_e = v$, where $k_e = \min\{k \in \mathbb{N}\ :\ k > 1$ **and** $-b + k \times (b - a)\ /\ v + a \geq 0\}$	$[(f(a) + c)\ /\ 2.0 + (f(b) + c)\ /\ 2.0$ $+ \sum_{i=1}^{v-1} (f(a - a \times i/v + b \times i/v) + c)]$ $\times (b/v - a/v)$
P'_5: (1,3,4,5,17)	$v \geq 1$ **and** $a = b$	0
P'_6: (1,2,17)	$v < 1$	undefined

In the last step, $c \times (a - b) = c \times |b - a|$ because the path condition is $a > b$. Hence, relation (1) holds for $(P_3,\ P'_3)$. Proceeding this way, all the five combinations can be verified. Thus, the program *Area* satisfies the selected metamorphic relation for any valid input.

Similarly, we can prove the following metamorphic relations that correspond to the mathematical formulas $\int_a^b [f_1(x)\ +\ f_2(x)]\,dx\ =\ \int_a^b f_1(x)\,dx\ +\ \int_a^b f_2(x)\,dx$ and $\int_a^b \lambda\, f(x)\,dx = \lambda \int_a^b f(x)\,dx$, respectively:

$$\textit{Area}\,(g,\ a,\ b,\ v) = \textit{Area}\,(f_1,\ a,\ b,\ v) + \textit{Area}\,(f_2,\ a,\ b,\ v) \qquad (2)$$

where $g(x) = f_1(x) + f_2(x)$ and $v \geq 1$.

$$\textit{Area}\,(g,\ a,\ b,\ v) = \lambda\,\textit{Area}\,(f,\ a,\ b,\ v) \qquad (3)$$

where $g(x) = \lambda\, f(x)$ for some constant λ and $v \geq 1$.

2.7 Extrapolation of Correctness for Concrete Test Cases

Geller [28] proposed a strategy to use test data as an aid in proving program correctness. The basic idea is to prove a program in two steps: First, show that the program meets its specification for a sample test case. Then, prove the computed output of the program and the function defined by the specification "are perturbed in the same fashion" as the values of the input are perturbed (which can be considered as a special kind of metamorphic relation.) In this way, the program can satisfy the *generalization assertion*, which generalizes "from the value produced by the program for some given test value to a larger domain." In the conclusion section, Geller pointed out that "A great deal of work remains to be done in the area of using test data to prove program correctness. ... Many additional theorems could be proven, ... to be used in generalizing from specific test data to a larger domain."

We find that metamorphic relations are ideal properties to support the generalization assertion. Consider the program *Area*, for example. We have proven relations (1), (2), and (3) in Section 2.6. To further verify the program, let $f_1,\ f_2,\ \ldots,\ f_n$ be n functions of x, where $n \geq 1$. We can test the program *Area* using n concrete test cases $(f_1,\ a_0,\ b_0,\ v_0)$, $(f_2,\ a_0,\ b_0,\ v_0)$, $\ldots$, $(f_n,\ a_0,\ b_0,\ v_0)$, where a_0, b_0, and v_0 are constants. Suppose the program successfully passes these concrete test cases (otherwise a failure is detected and the testing can stop.) Let g be any function that can be expressed as a linear combination of $f_1,\ f_2,\ \ldots,\ f_n$ and a constant, that is, $g(x) = \lambda_1\, f_1(x) + \lambda_2\, f_2(x) + \ldots + \lambda_n\, f_n(x) + \lambda_{n+1}$, where $\lambda_1,\ \lambda_2,\ \ldots,\ \lambda_{n+1}$ are real constants. Then, according to the three relations we have proven, the output of the program for the test case $(g,\ a_0,\ b_0,\ v_0)$ must satisfy the following relation:

$$\begin{aligned} & \textit{Area}\,(g,\ a_0,\ b_0,\ v_0) \\ = \ & \lambda_1\,\textit{Area}\,(f_1,\ a_0,\ b_0,\ v_0) + \lambda_2\,\textit{Area}\,(f_2,\ a_0,\ b_0,\ v_0) \\ & + \ldots + \lambda_n\,\textit{Area}\,(f_n,\ a_0,\ b_0,\ v_0) + \lambda_{n+1}\,|b_0 - a_0| \end{aligned}$$

If the results of the program are correct for the test cases $f_1(x)$, $f_2(x)$, $\ldots$, $f_n(x)$, it implies that the program is also correct for $g(x)$. Since λ_1, λ_2, $\ldots$, λ_{n+1} can take *any* real values, our method has indeed established the correctness of the results for *infinitely* many inputs (that is, g) based on *finitely* many concrete test cases (that is, f_1, f_2, $\ldots$, f_n.)

3 A Case Study on the Siemens Program *replace*

We have implemented our method to verify programs written in C. The implemented system consists of two major components: The first component supports symbolic executions and the second supports the verification of metamorphic relations. For the first component, the techniques and tools introduced by Sen et al. [52] have been employed to systematically explore the symbolic execution tree using a depth-first search strategy. We found this to be an effective approach to dealing with loops and arrays. Where infinite loops may be encountered, users can set an upper bound for the depth of the search so that the latter can always terminate. For the second component, users need to specify the expected metamorphic relations in terms of embedded C code in the test driver. For example, if the program under test is a function named int *testme*(int a, int b), where a and b are integer parameters, and the expected metamorphic relation is *testme*(a, b) == *testme*(b, a), then the main body of the code that users need to write to specify the metamorphic relation is as follows:

```
generateSymbolicInput(a);
generateSymbolicInput(b);
verify(testme(a, b) == testme(b,a));
```

3.1 The Subject Program

We studied the *Siemens suite of programs* [41][3] as candidate subjects for our empirical evaluation. The Siemens suite consists of seven programs, namely, *print_tokens*, *print_tokens2*, *replace*, *schedule*, *schedule2*, *tcas*, and *tot_info*. As pointed out by Liu et al. [49], the *replace* program is the largest and most complex among the seven programs, and covers the most varieties of logic errors. Thus, we conducted an empirical evaluation of our techniques using the *replace* program. This program has 563 lines of C code (or 512 lines of C code excluding blanks and comments) and 20 functions. It has 32 faulty versions in the suite.

The *replace* program performs regular expression matching and substitutions. It receives three input parameters. The first parameter is a regular expression. Note that it is not a regular expression as usually defined in theoretical computer science, but the Unix version with typographic and extended properties. The second parameter is a string of text, and the third is another string of text. For any substring(s) of the third parameter that matches (that is, is an instance of) the regular expression, the program will replace it by the second parameter. For example, if the first parameter is 'ab[cde]f*[^gh]i', the second is 'xyz', and the third is '8abdffffxiyes', then the program will produce the output '8xyzyes'. This is because the substring 'abdffffxi' matches the regular expression 'ab[cde]f*[^gh]i' and is therefore replaced by 'xyz'.

3.2 The Metamorphic Relations

Our method verifies the subject program through metamorphic relations. We appreciate that it is unlikely for a single MR to detect all the faults. We have therefore identified four MRs that are quite different from one another with a view to detecting various faults. 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. They are described below.

MR1: The intuition behind MR1 is that, given the text 'ab', replacing 'a' with 'x' is equivalent to replacing non-'b' with 'x'. The syntax of the former is *replace* 'a' 'x' 'ab' while the syntax of the latter is *replace* '[^b]' 'x' 'ab'. For both executions, the output will be 'xb'. The following description gives more details of MR1.

Let pat_1 be a simple regular expression that represents (1) a single character such as 'a', or (2) a range of characters in the form of '[x-z]'. The end-of-line symbol '$' can be added to the end of a regular expression. For the initial test case, the first parameter is pat_1; the second is *char*, which is an arbitrary character; and the third is $\mathit{text}_1\mathit{text}_2$ (concatenation of text_1 and text_2), where text_1 and text_2 are strings such that

- text_1 or text_2 matches pat_1, and
- if text_1 (text_2, respectively) matches pat_1, then text_2 (text_1, respectively) or any of its nonempty substrings does not match pat_1.

Suppose text_1 matches pat_1. Then, for the follow-up test case, the first parameter is '[^text_2]'[4] The second and third parameters remain unchanged. The expected relation between the initial and follow-up executions is that their outputs should be identical.

When text_2 matches pat_1, the treatment is similar.

MR2: Let *char* be a single character. MR2 observes that the expressions '*char*' and '[*char*]' are equivalent with the exception of a few special cases (for example, '?' is not equivalent to '[?]' — the former is a wildcard but the latter represents the question mark.) Hence, replacing '*char*' by a string is equivalent to replacing '[*char*]' by the same string. For example, the outputs of *replace* 'a' 'x' 'abc' and *replace* '[a]' 'x' 'abc' should be identical; the outputs of *replace* 'a*' 'x' 'aa' and *replace* '[a]*' 'x' 'aa' should be identical; and, for the wildcard character '?', the outputs of *replace* '?*' 'x' 'aa' and *replace* '[^]*' 'x' 'aa' should be identical.

[3] The Siemens suite of programs used in our experiments was downloaded from http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/.

[4] Some special characters, such as '@', will have different meanings in text_2. For instance, if text_2 consists of '@t', it means the character '@' followed by the character 't'. However, if text_2 consists of '@@t', it means the character '@' followed by a tab.

MR3 makes use of a particular regular expression `'?*'`, where `'?'` is a wildcard character and `'*'` means the Kleene closure. Therefore, `'?*'` will match the entire input string. For instance, the outputs of *replace* '?*' 'x' 'ab' and *replace* 'ab' 'x' 'ab' should be identical. Further, the outputs of *replace* '?*' 'x' 'ab' and *replace* '?*' 'y' 'ab' should not be equal because `'x'` $\neq$ `'y'`.

MR4 makes use of equivalence properties of regular expressions that involve square brackets. For example, `'[ABC]'` is equivalent to any of its permutations, such as `'[BCA]'` and `'[CBA]'`; `'[2-5]'` is equivalent to `'[2345]'`; `'[0-z]'` is equivalent to `'[0123 ... xyz]'`, that is, the set of all characters whose ASCII values are between `'0'` and `'z'`, inclusive; *replace* '[0–z]' 'x' '09' is equivalent to *replace* '[0–9]' 'x' '09'; and so on.

3.3 Results of Experiments

We have made the following minor changes to the source code of the *replace* program for the purpose of facilitating the experiments: (1) We rename the original "main" function so that it is no longer the main function of the program. This is because the main function of the updated program is the test driver and the original main function is called by the test driver. (2) We bypass the *getline* function, which reads input from the keyboard. We replace it by a function that generates symbolic input. (3) Similarly, we also revise the output function *fputc* so that it can print symbolic output. In the experiments, the symbolic inputs we generate are fixed-sized arrays that represent strings of characters.

The results of the experiments are summarized in Table 7, where version 0 is the base program and versions 1 to 32 are faulty versions (created by Siemens researchers by manual injection of faults into the base program [41].) If an MR detects a failure for a version, then we place the statistical data regarding the source code and the failure report in the corresponding table cell. The experimental results will be explained in more detail in Section 3.4.

Table 7 shows that, out of the 32 faulty versions, MR1 detected failures for 18 versions, and MR2, MR3, and MR4 detected failures for 16, 8, and 10 versions, respectively.

For version 8, its failure can be detected directly during the initial execution. This is because when the first input parameter is in the form of a letter followed by `'**'` (such as `'b**'`), version 8 will terminate abnormally. A failure is therefore revealed without the need to refer to any metamorphic relation.

For version 19, the fault lies in the *getline* function, which reads input from the keyboard. Since we have replaced this function with another one in our experiments to generate symbolic input, any fault in *getline* cannot be detected.

For version 15, no failure was detected using the four metamorphic relations we defined. The difference between version 15 and version 0 is that when the first input parameter is `''`, the former will consider it to be an empty string, whereas the latter will take it as an illegal pattern and terminate. Such failures due to the mishandling of degenerate inputs cannot be revealed in the absence of a design specification and are unrelated to the metamorphic relations in question.

Table 7 shows that different MRs demonstrate different fault-detection capabilities: MR1 detects the largest number of faults, and MR3 the smallest. However, we also observe that different MRs are complementary to one another. For example, although the overall performance of MR3 was the worst, it is the only MR violated by versions 3 and 4. We can see from Table 7 that, apart from version 8, each MR has detected violations that have not been detected by others, and none of the faulty versions violate all the MRs.

The above observations suggest that even the best MR may not be able to uncover all faults and, therefore, we should try to employ more than one MR for verifying programs.

On the other hand, as different MRs have different fault-detection capabilities, it is an important and challenging task to investigate how to prioritize MRs, especially if there are many of them. It is beyond the scope of this paper to study MR prioritization. Nevertheless, we note that if the initial and follow-up executions are very different (say, exercising very different paths), the chance of violating an MR will be relatively higher. This observation is not only consistent with our previous results reported in [16], but also consistent with the intuition about the effectiveness of coverage testing: The higher the coverage is, the better the fault-detection capability will be.

Apart from the perspective of fault-detection capabilities, another perspective toward the prioritization of MRs is that different users may rank different MRs higher according to their needs. Some users may use only part of the functionality offered by the program or a proper subset of the input domain. Some MRs may therefore be essential to some users but less important to others. Consider a program calculating the sine function, for instance. The periodic property $\sin(2n\pi + x) = \sin x$ is essential for electrical engineers, as they often use the program to compute the waveform of the AC circuit, which exhibit periodicity. These engineers would therefore select the periodic property for the entire real domain as an important MR to verify the program. On the other hand, surveyors may not use the periodic property often. Instead, they may be more interested in other properties of sine, such as $\sin(\pi - x) = \sin x$. Such a prioritization approach can be considered to be based on the perspective of reliability, where usage profiles are taken into consideration.

3.4 Experience of Applying the Debugging Technique

In our verification system, the source code of any program under test is instrumented using a program instrumentor prior to compilation, so that an execution trace can be collected. When a violation of a metamorphic relation occurs, a failure report will be generated in two steps. First, details of the initial and follow-up executions are recorded. Then, diagnostic details are added.

Fig. 5 shows a typical failure report, generated by our debugger for version 14 of the *replace* program verified against MR1. In line 370 of version 14, a subclause "&&

TABLE 7
Results of Experiments

Each filled table cell indicates that the corresponding version detects the violation of the corresponding MR.
Format of each filled table cell: no. of highlighted lines of source code / total no. of lines of source code;
no. of highlighted rows in failure report / total no. of rows in failure report

version	MR1	MR2	MR3	MR4
0				
1	58/512=11%; 35/338=10%	20/512=4%; 11/201=5%		42/512=8%; 32/993=3%
2	58/508=11%; 37/340=11%			42/508=8%; 32/993=3%
3			63/512=12%; 46/350=13%	
4			10/512=2%; 5/108=5%	
5	45/512=9%; 43/319=13%			45/512=9%; 59/629=9%
6	63/512=12%; 56/399=14%			63/512=12%; 56/932=6%
7		20/512=4%; 13/167=8%	20/512=4%; 13/167=8%	
8	28/512=5%; 16/315=5%	28/512=5%; 16/315=5%	28/512=5%; 16/315=5%	28/512=5%; 16/315=5%
9	42/511=8%; 24/288=8%			42/511=8%; 24/365=7%
10				20/512=4%; 180/248=73%
11	42/512=8%; 24/288=8%			42/512=8%; 24/365=7%
12				26/512=5%; 128/490=26%
13		56/515=11%; 44/463=10%	56/515=11%; 44/406=11%	
14	52/512=10%; 31/231=13%			
15				
16		20/512=4%; 13/167=8%	20/512=4%; 13/167=8%	
17	33/512=6%; 19/310=6%	33/512=6%; 19/359=5%		
18	58/512=11%; 32/284=11%	28/512=5%; 30/187=16%		
19				
20	33/512=6%; 19/298=6%	33/512=6%; 19/347=5%		
21		12/512=2%; 6/168=4%		
22	44/511=9%; 26/262=10%	66/511=13%; 49/520=9%		
23	33/512=6%; 19/380=5%	33/512=6%; 19/429=4%		
24	55/511=11%; 51/447=11%			
25	53/512=10%; 49/437=11%			
26	44/512=9%; 68/474=14%	67/512=13%; 95/794=12%		
27	20/512=4%; 10/187=5%		25/512=5%; 11/53=21%	
28		20/512=4%; 13/121=11%		
29		20/512=4%; 13/222=6%		
30		20/512=4%; 13/188=7%	20/512=4%; 13/167=8%	
31	58/512=11%; 33/327=10%	28/512=5%; 30/187=16%		
32				63/512=12%; 39/619=6%

(!locate(lin[*i], pat, j+1))” that should have been included in the if-condition is omitted by the programmer. We have run this version on the 5,542 test cases provided by the Siemens suite, compared the outputs against those of the base program (version 0), and found that 137 out of the 5,542 outputs are different.

In Fig. 5, the symbolic input vector for the initial execution is printed in rows 1, 2, and 3, where each of `v1`, `v2`, `v3`, and `v4` represents a single character, and `v1` is the first parameter, `v2` is the second, and the concatenation of `v3` and `v4` is the third. Because of space limit, certain rows have been omitted. Each “symbolic output” printed in the report corresponds to a single character. For instance, rows 105 and 130 report that the first output character has an ASCII value equal to that of input `v2`, and the second output character has an ASCII value equal to that of input `v4`. (If a symbolic output expression is “`4 * v2 + 5 * v3`”, for example, it means that the output character has an ASCII value equal to 4 × `v2` + 5 × `v3`, where `v2` and `v3` represent the ASCII values of the input characters.) An example of a concrete input vector of the initial execution (referred to as “1st run”) is given in row 276, where `v1`, `v2`, `v3`, and `v4` take the concrete values of `'@'`, `'A'`, `'@'`, and `'B'`, respectively. The resultant concrete output is given in row 277, where the first output character is `'A'` (which corresponds to `v2`) and the second output character is `'B'` (which corresponds to `v4`.)

Rows 136, 137, and 138 print the symbolic values of the input vector for the follow-up execution (referred to as “2nd run.”) The first parameter is `'[^v4]'`. If `v4` takes a concrete value of `'b'`, for instance, then this parameter will be `'[^b]'`. The second and third parameters remain the same as the initial execution. The symbolic values of the first and second output characters are printed in rows 229 and 263, respectively. Row 264 reports that a failure is detected. Rows 266 and 267 further identify the trigger of failure to be “`1 * v4 + -1 * v2 != 0`”, that is, a failure will occur when `v4` ≠ `v2`. It means that, when `v4` = `v2`, the initial and follow-up execution paths will not violate the MR. This diagnostic message therefore reminds users to pay attention to the condition `v4` ≠ `v2` (which is indeed the condition under which the output goes wrong.) Row 278 gives an example of a concrete input vector for the follow-up execution, and this is indeed a failure-causing input. The system further finds that the initial execution path has not been involved with any other failure, but the follow-up execution path has appeared in another failure, as reported in rows 280 and 281. The system therefore

```
1   [symbolic input of 1st run] v1
2   [symbolic input of 1st run] v2
3   [symbolic input of 1st run] v3 v4
4   [trace] entry to main(), line 523
... ...
103 [trace] false branch taken, line 468
104 [pc]    1 * v2  !=  -1
105 [symbolic output]   1 * v2 + 0
... ...
130 [symbolic output]   1 * v4 + 0
... ...
134 [trace] return from main(), line 554
135 -------- end of 1st run --------
136 [symbolic input of 2nd run] [^v4]
137 [symbolic input of 2nd run] v2
138 [symbolic input of 2nd run] v3 v4
139 [trace] entry to main(), line 523
140 [trace] false branch taken, line 527
141 [trace] entry to getpat(), line 249
... ...
209 [trace] switch, line 347
210 [trace] case NCCL, line 369
211 [trace] true branch taken, line 370
212 [pc]    1 * v3  !=  10
213 [trace] true branch taken, line 378
214 [trace] return from omatch(), line 384
... ...
228 [trace] false branch taken, line 468
229 [symbolic output]   1 * v2 + 0
230 [debugging diagnosis] output has
    been normal up to this point
231 [trace] false branch taken, line 467
232 [trace] return from putsub(), line 479
233 [trace] false branch taken, line 498
234 [trace] true branch taken, line 491
235 [trace] entry to amatch(), line 422
236 [trace] true branch taken, line 427
237 [trace] false branch taken, line 428
238 [trace] entry to omatch(), line 332
239 [trace] false branch taken, line 337
240 [trace] entry to in_pat_set(), line 181
241 [trace] return from in_pat_set(),
    line 182
```

```
242 [trace] false branch taken, line 341
243 [trace] switch, line 347
244 [trace] case NCCL, line 369
245 [trace] true branch taken, line 370
246 [pc]    1 * v4  !=  10
247 [trace] true branch taken, line 378
248 [trace] return from omatch(), line 384
249 [trace] false branch taken, line 448
250 [trace] entry to patsize(), line 391
251 [trace] entry to in_pat_set(), line 181
252 [trace] return from in_pat_set(), line 182
253 [trace] false branch taken, line 393
254 [trace] switch, line 397
255 [trace] case NCCL, line 404
256 [trace] return from patsize(), line 413
257 [trace] false branch taken, line 427
258 [trace] return from amatch(), line 454
259 [trace] true branch taken, line 494
260 [trace] entry to putsub(), line 462
261 [trace] true branch taken, line 467
262 [trace] false branch taken, line 468
263 [symbolic output]   1 * v2 + 0
264 [debugging diagnosis] failure is detected for the above output
265 [debugging diagnosis] expected output is    1 * v4 + 0
266 [debugging diagnosis] trigger of failure is as follows:
267     1 * v4  + -1 * v2  !=  0
268 [trace] false branch taken, line 467
269 [trace] return from putsub(), line 479
270 [trace] false branch taken, line 498
271 [trace] false branch taken, line 491
272 [trace] return from subline(), line 504
273 [trace] return from change(), line 518
274 [trace] return from main(), line 554
275 -------- end of 2nd run --------
276 [example input of 1st run]  '@' 'A' '@B'
277 [example output of 1st run] 'AB'
278 [example input of 2nd run]  '[^B]' 'A' '@B'
279 [example output of 2nd run] 'AA'
280 [frequency] path of 1st run occurs in violations of MR:
    1 time(s)
281 [frequency] path of 2nd run occurs in violations of MR:
    2 time(s)
282 [current debugging focus] 2nd run
```

Fig. 5. Failure Report for Version 14 of the *replace* Program (where certain rows have been omitted owing to space limit)

suggests users to pay attention to the follow-up execution, as printed in row 282. The system further identifies that the output has been normal up to row 229, but goes wrong in row 263. Therefore, row 230 reports that "`output has been normal up to this point`", and rows 231 to 263 are highlighted because they constitute the portion of the execution directly leading to the failure. In fact, the failure is caused by the event reported in row 245: "`[trace] true branch taken, line 370`" — a correct execution should take a false instead of a true branch at this decision point. Row 265 further reports that the expected symbolic value of the output in row 263 should be "`1 * v4 + 0`", which is the symbolic value produced by the initial execution (in row 130.)

Based on the highlighted debugging information in Fig. 5, our debugger further highlights a total of 52 lines of source code for inspection. These lines of code are not shown in this paper. Version 14 has a total of 512 lines of source code excluding blanks and comments. The number of highlighted lines of source code can be greater than the number of highlighted rows in the failure report as multiple lines of source code may be executed (and highlighted) between two rows in the failure report. We can go on to employ *program slicing* techniques (see, for example, [2], [46]) to further exclude statements having no impact on the output value reported in row 263. In this way, the focus of debugging will be further narrowed down. Such a program slicing component, however, has not been incorporated into our current system owing to resource limits and also because this component is not the focus of our present research project.

It has been found that for each of the 52 failures reported in Table 7, the user can always identify the root cause of the failure using the highlighted debugging information of the failure report. Furthermore, for each of the 52 failures, two different percentages are computed and reported in Table 7 as indicators of the debugging effort. The first is the *source line percentage* (SLP), which represents the number of highlighted lines of source code divided by the total number of lines of the source code. The second is the *failure report percentage* (FRP), which represents the number of highlighted rows in the failure report divided by the total number of rows in the failure report. It should be noted that the number of rows of a failure report varies, depending on the user's configuration that specifies the types of information to be printed. In any case, for a nontrivial program, the main content of a failure report is the execution trace (that is, the rows starting with "[trace].") For ease of comparison, therefore, we count only this type of row when calculating FRP. Hence, for the failure report shown in Fig. 5, its total number of rows is counted as 231, of which 31 are highlighted (namely, rows 231 to 245 and rows 247 to 262.)

To summarize the 52 filled cells of Table 7, the SLP values range from 2 percent (minimum) to 13 percent (maximum) with a mean of 7 percent. The FRP values range from 3 percent (minimum) to 73 percent (maximum) with a mean of 10 percent. The highest FRP (73 percent) is reached when version 10 is verified against MR4. We find that, although this percentage is high, the highlighted rows are indeed a genuine cause-effect chain: The fault in the program has caused many additional iterations of a loop body and, therefore, a long list of execution traces within the loop body have been highlighted. The number of source lines highlighted, on the other hand, is quite small (SLP = 4%.) The second largest FRP (26 percent) occurs when version 12 is verified against MR4. This is also due to iterations of a loop. In fact, the 128 rows of highlighted execution traces involve only 16 unique rows, and the number of source lines highlighted is also small (SLP = 5%.) It can be concluded that the experimental results shown in Table 7 demonstrate that our debugging technique can effectively reduce users' debugging effort.

4 Limitations and Potential Remedies

The underlying toolset that supports semi-proving is global symbolic evaluation and symbolic execution. We recognize that fully automated global symbolic evaluation and constraint solving may not necessarily be realized for any arbitrary program, especially for those involving complex loops, arrays, or pointers. In such situations, we can more effectively conduct semi-proving on a selected set of paths to achieve specific coverage criteria, or to verify specific critical paths. In this situation, semi-proving becomes a symbolic-testing approach, that is, testing with symbolic inputs. Note that, in this situation, we will need to set an upper bound for the search in the symbolic execution tree. The value of such an upper bound depends on the testing resources available. This value directly affects the cost-effectiveness of the method and may be difficult to decide. Setting such an upper bound will also limit the fault-detection capability of our method when the fault can only be detected at a deeper level of the symbolic execution tree. This is, however, a general problem faced by any verification method based on symbolic executions.

Another limitation of symbolic-execution-based approaches is the complexity of path conditions. To alleviate this problem, a dynamic approach that combines both symbolic and concrete executions has been developed (see Godefroid et al. [32] and Sen et al. [52].) When some symbolic expressions cannot be handled by the constraint solver, the CUTE tool [52], for instance, will replace a symbolic value in the expression with a concrete value so that the complexity of the expression can always be under control. This approach has also been applied to the testing of dynamic Web applications written in PHP [4]. A weakness of this strategy is that it sacrifices generality for scalability.

When there are a large number of paths to verify, the efficiency of symbolic-execution-based approaches is also a concern. There are, however, algorithms and tools that tackle such tasks more efficiently. The Java PathFinder model checker [1], [44], for example, uses a backtracking algorithm to traverse the symbolic execution tree instead of starting from scratch for every symbolic execution.

To achieve higher scalability for large software applications, it has been proposed by Godefroid [31] to use summaries, generated by symbolic execution, to describe the behavior of each individual function of the program under test. Whenever a function is called, the instrumented program under test will check whether a summary for that function is already available for the current calling context. This algorithm is functionally equivalent to DART [32] but can be much more efficient for large applications.

We would like to point out that, although symbolic execution can provide higher confidence of the correctness of programs than concrete executions, it may need to have caveats when used to prove program properties. To verify a division function, for instance, a metamorphic relation can be identified as $(a/b) \times (b/c) = (a/c)$. If "b==0" does not appear in path conditions of the division function, some symbolic executors may "prove" that this metamorphic relation is satisfied without warning the user that b must take a nonzero value. Our method should therefore be used in conjunction with other techniques that check safety properties, such as those for detecting data overflow/underflow and memory leak (see [30], for instance.)

Another limitation of our method is the availability of metamorphic relations. The first category of programs amenable to metamorphic relations that one can think of is, of course, numerical programs. However, metamorphic relations also widely exist for many programs that do not involve numerical inputs. The *replace* program in our case study, for instance, involves only strings of characters as inputs. Zhou et al. [60] have identified a number of metamorphic relations to test Web search engines. In pervasive computing [10] and Web services [11], metamorphic relations have also been identified and employed for software testing.

For the purpose of debugging, a limitation of our method is that multiple execution paths are involved. While heuristic approaches do exist to help identify genuine failure paths, users may have to inspect all the execution paths in worst case scenarios.

Another limitation in debugging is the identification of the trigger of the failure, which is obtained by calculating the difference between the two conditions MFCC and MPC. The computation of the difference is nontrivial and sometimes may not be done automatically. It should also be noted that different MRs may generate different MFCCs and MPCs. Even for the same MR, there can be more than one MPC. As a result, different triggers may be generated for the same fault. On the other hand, each of these triggers will provide clues for the discovery of the nature of the fault. It is worthwhile to conduct future research on the usefulness of various triggers.

5 Comparisons with Related Work

As semi-proving is an integrated method for testing, proving, and debugging, we will compare it with related work along the lines of these three topics.

5.1 Testing

While semi-proving finds its basis from metamorphic testing, the former has incorporated many new features that have never been considered by the latter.

Semi-proving uses symbolic inputs and hence entails nontrivial effort in its implementation, whereas metamorphic testing uses concrete inputs and is therefore much easier to implement. Semi-proving, however, has unique advantages. First, not all inputs that exercise a failure path may necessarily cause a failure. Among all the inputs that exercise the failure paths in the program $\overline{Med}$ discussed in Sections 2.4 and 2.5, for instance, only those satisfying the **trigger** cause a failure. Whether or not a concrete test case meets the trigger condition is purely by chance. On the other hand, semi-proving guarantees the detection of the failure. Second, if the program under test is correct, metamorphic testing with concrete inputs can only demonstrate that the program satisfies the metamorphic relation for the finite set of tested inputs. Semi-proving, on the other hand, can demonstrate the satisfaction of the metamorphic relation for a much larger set of inputs, which may be an infinite set. This gives a higher confidence. Furthermore, semi-proving can be further combined with conventional testing to extrapolate the correctness of the program to related untested inputs. Third, when a failure is detected, semi-proving will provide diagnostic information for debugging in terms of constraint expressions, but metamorphic testing does not have this capability.

5.2 Proving

A method has been developed by Yorsh et al. [57] to combine testing, abstraction, and theorem 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 proven.

A common ground between this method and ours is that both methods attempt to verify necessary properties for program correctness. The properties of interest, however, are very different between the two approaches. Yorsh et al.'s method [57] 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* [48], [49]. 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 are therefore complementary to each other — this has been discussed previously. Second, the objectives of the two methods are different. Yorsh et al.'s method [57] "is oriented towards finding a proof rather than detecting real errors," and "does not distinguish between a false error and a real error." On the other hand, semi-proving is intended for both proving properties and detecting errors, and any error detected by semi-proving is a real error. There is no false error in semi-proving. Furthermore, although both Yorsh et al.'s method and semi-proving combine testing and proving, this is done in different senses. The former enhances testing (with concrete inputs) to the proof of safety properties. Semi-proving extrapolates testing (with concrete or symbolic inputs) to the correctness of the program for related untested inputs.

Gulavani et al. [35] proposed an algorithm "SYNERGY," which combines testing and proving to check program properties. SYNERGY unifies the ideas of counterexample-guided model checking, directed testing [32], and partition refinement. According to Gulavani et al. [35], the SYNERGY algorithm can be more efficient in constructing proofs of correctness for programs with the "diamond" structure of if-then-else statements, compared with the algorithm used in DART [32].[5] This is because the goal of SYNERGY is different from that of other testing methods: It does not attempt to traverse the execution tree; instead, it attempts to cover all *abstract states* (equivalence classes.) Therefore, SYNERGY does not need to enumerate all the paths.

Our method, on the other hand, does not involve abstraction. It adopts symbolic-analysis and symbolic execution techniques, and more information is generated for debugging when a failure is detected. We note that metamorphic relations are a type of property different from safety properties conventionally discussed in the literature, and MRs involve outputs of multiple executions of a program under test. As program output is affected by almost every part of its code, the efficiency of applying abstraction to the verification of metamorphic relations is unclear, but worth investigation in future research.

5.3 Debugging

Zeller and Hildebrandt [59] proposed a *Delta Debugging* algorithm that transforms a failure-causing input into a minimal form that can still fail the program. This is done by continuously narrowing down the difference between failure-causing and non-failure-causing inputs. Zeller [58] further developed the Delta Debugging method by examining "what's going on inside the program" during the execution. He considered the failed execution as a sequence of program states, and only part of the variables and values in some of the states are relevant to the failure. He proposed isolating these relevant variables and values by continuously narrowing the difference in program states in successful and failed executions. This narrowing process

[5] The time complexity of the depth-first search algorithm used in our implementation is similar to that of DART.

was conducted by altering the runtime program states and then assessing the outcomes of the altered executions.

Although our debugging approach also compares successful and failed executions, it is very different from the above techniques. Suppose we are given a program $P(x, y)$ and two test cases $t_1 : (x = 3, y = 5)$ and $t_2 : (x = 3, y = 2)$. Suppose P computes correctly for t_1 but fails for t_2. Zeller's method [58] would isolate $y = 2$ as relevant to the failure because the program fails after altering y from 5 to 2. We observe, however, that concrete values may not necessarily show the root cause of a failure in many situations. The failure of program P, for example, may not be relevant only to the variable y and the value "2," but pertinent to some relation between both variables x and y, such as when $x > y$. In this way, test cases like $t_3 : (x = 6, y = 5)$ will also show a failure. Our approach therefore identifies a set of failure-causing inputs in terms of constraint expressions that provide additional information about the characteristics of the defect. Another difference is that their methods assume the existence of a "testing function" that can tell whether a test result is "pass," "fail," or "unresolved." This testing function is similar to an automated oracle. For many problems, however, such a function is not available. For instance, the outputs of a program that computes the sine function cannot be categorized into "pass" or "fail" automatically unless there is a testing function that *correctly* computes the sine function. On the other hand, by making reference to metamorphic relations, our method can be applied in such situations.

He and Gupta [39] introduced an approach to both locating and correcting faulty statements in a program under test. The approach combines ideas from correctness proving and software testing to locate a likely erroneous statement and then correct it. It assumes that a correct specification of the program under test is given in terms of preconditions and postconditions. It also assumes that only one statement in the program under test is at fault. Using the concept of *path-based weakest precondition*, the notions of a *hypothesized program state* and an *actual program state* at every point along the failure path (execution trace) are defined. He and Gupta's algorithm [39] traverses the failure path and compares the states at each point to detect *evidence* for a likely faulty statement. Such "evidence" will emerge if a predicate representing the actual program states is less restrictive than the predicate representing the hypothesized program states. The algorithm then generates a modification to the likely faulty statement. The modified program is then tested using all existing test cases.

The above method requires that the program under test contain "a single error statement" and that the preconditions and postconditions be given in terms of first order predicate logic. The experiments conducted by He and Gupta [39] were on a small scale, with a limited number of error types. We observe that, as metamorphic relations can be used as a special type of postcondition, there is potential for the above method to be combined with semi-proving to locate and correct faults. This will be an interesting topic for future research.

Our debugging technique is also different from the approach developed by Jones et al. [43]. Their approach is to design and assign a probability to each statement of the program to indicate how likely it is that this statement is at fault. This is a probabilistic approach and does not look into the cause of the failure. Our approach, on the other hand, looks into the cause-effect chain that leads to the failure.

6 Conclusion

We have presented an integrated method for proving, testing, and debugging. First, it proves that the program satisfies selected program properties (that is, metamorphic relations) throughout the entire input domain or for a subset of it. When combined with concrete test cases, our method may also enable us to extrapolate from the correctness of a program for the concrete test cases to the correctness of the program for related untested inputs. Second, our method is also an automatic symbolic-testing technique. It can be used to test selected program paths. It employs both black-box (functional) knowledge from the problem domain for selecting metamorphic relations and white-box (structural) knowledge from program code for symbolic execution. As a result, subtle faults in software testing, such as the missing path errors, can be better tackled. Third, our method also supports automatic debugging through the identification of constraints for failure-causing inputs.

The implementation and automation of semi-proving is also relatively easier than conventional program proving techniques. This is because metamorphic relations are weaker than program correctness and hence they can be easier to prove.

For future research, apart from the topics highlighted in relevant places in Sections 4 and 5, we will also study the effectiveness of different metamorphic relations for different types of faults.

7 Acknowledgments

The authors are grateful to Willem Visser for his helps and discussions on symbolic execution techniques using the Java PathFinder model checker [1], [44]. They are also grateful to Giovanni Denaro and his group for providing the symbolic execution tool SAVE [21]. They would like to thank Joxan Jaffar and Roland Yap for providing a constraint solver [42]. They would also like to thank Bernhard Scholz and Phyllis Frankl for their information and discussions on symbolic-evaluation techniques.

References

[1] Java PathFinder Home Page. http://babelfish.arc.nasa.gov/trac/jpf, 2010.

[2] 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, 1995.

[3] P.E. Ammann and J.C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[4] 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, 2008.

[5] L. Baresi and M. Young, "Test oracles," Technical Report CIS-TR01-02, Department of Computer and Information Science, University of Oregon, Eugene, OR, 2001.

[6] B. Beizer, *Software Testing Techniques*. New York, NY: Van Nostrand Reinhold, 1990.

[7] M. Blum and S. Kannan, "Designing programs that check their work," *Proceedings of the 31st Annual ACM Symposium on Theory of Computing* (*STOC '89*), pp. 86–97. New York, NY: ACM, 1989. Also *Journal of the ACM*, vol. 42, no. 1, pp. 269–291, 1995.

[8] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing / 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, 1990. Also *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 549–595, 1993.

[9] B. Burgstaller, B. Scholz, and J. Blieberger, *Symbolic Analysis of Imperative Programming Languages*, Lecture Notes in Computer Science, vol. 4228, pp. 172–194. Berlin, Germany: Springer, 2006.

[10] W.K. Chan, T.Y. Chen, H. Lu, T.H. Tse, and S.S. Yau, "Integration testing of context-sensitive middleware-based applications: a metamorphic approach," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 5, pp. 677–703, 2006.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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* (*COMPSAC 2002*), pp. 327–333. Los Alamitos, CA: IEEE Computer Society, 2002.

[16] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering* (*JIISIC 2004*), pp. 569–583. Madrid, Spain: Polytechnic University of Madrid, 2004.

[17] 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*), pp. 191–195. New York, NY: ACM, 2002.

[18] T.Y. Chen, T.H. Tse, and Z.Q. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, 2003.

[19] L.A. Clarke and D.J. Richardson, "Symbolic evaluation methods: implementations and applications," *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (editors), pp. 65–102. Amsterdam, The Netherlands: Elsevier, 1981.

[20] W.J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions*. Englewood Cliffs, NJ: Prentice Hall, 1980.

[21] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè, "Using symbolic execution for verifying safety-critical systems," *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundation of Software Engineering* (*ESEC 2001/FSE-9*), pp. 142–151. New York, NY: ACM, 2001.

[22] R.A. DeMillo and A.J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[23] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Berlin, Germany: Springer, 2008.

[24] L.K. Dillon, "Using symbolic execution for verification of Ada tasking programs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 643–669, 1990.

[25] T. Fahringer and A. Požgaj, "P^3T+: a performance estimator for distributed and parallel programs," *Scientific Programming*, vol. 8, no. 2, pp. 73–93, 2000.

[26] T. Fahringer and B. Scholz, "A unified symbolic evaluation framework for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 11, pp. 1105–1125, 2000.

[27] T. Fahringer and B. Scholz, *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*, Lecture Notes in Computer Science, vol. 2628. Berlin, Germany: Springer, 2003.

[28] M. Geller, "Test data as an aid in proving program correctness," *Communications of the ACM*, vol. 21, no. 5, pp. 368–375, 1978.

[29] M.P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 1, pp. 85–122, 1995.

[30] P. Godefroid, M.Y. Levin, and D.A. Molnar, "Active property checking," *Proceedings of the 8th ACM International Conference on Embedded Software* (*EMSOFT 2008*), pp. 207–216. New York, NY: ACM, 2008.

[31] P. Godefroid, "Compositional dynamic test generation," *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2007*), pp. 47–54. New York, NY: ACM, 2007.

[32] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2005*), pp. 213–223. New York, NY: ACM, 2005.

[33] A. Gotlieb, "Exploiting symmetries to test programs," *Proceedings of the 14th International Symposium on Software Reliability Engineering* (*ISSRE 2003*), pp. 365–374. Los Alamitos, CA: IEEE Computer Society, 2003.

[34] A. Gotlieb and B. Botella, "Automated metamorphic testing," *Proceedings of the 27th Annual International Computer Software and Applications Conference* (*COMPSAC 2003*), pp. 34–40. Los Alamitos, CA: IEEE Computer Society, 2003.

[35] B.S. Gulavani, T.A. Henzinger, Y. Kannan, A.V. Nori, and S.K. Rajamani, "SYNERGY: a new algorithm for property checking," *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*), pp. 117–127. New York, NY: ACM, 2006.

[36] M.R. Haghighat and C.D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 4, pp. 477–518, 1996.

[37] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[38] M. Hall, Jr., *The Theory of Groups*. Providence, RI: AMS Chelsea, 1999.

[39] 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.

[40] W.E. Howden, "Reliability of the path analysis testing strategy," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 208–215, 1976.

[41] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," *Proceedings of the 16th International Conference on Software Engineering* (*ICSE '94*), pp. 191–200. Los Alamitos, CA: IEEE Computer Society, 1994.

[42] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap, "The CLP($\mathcal{R}$) language and system," *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 3, pp. 339–395, 1992.

[43] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pp. 467–477. New York, NY: ACM, 2002.

[44] S. Khurshid, C.S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS 2003*). Warsaw, Poland, 2003.

[45] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[46] B. Korel and J. Rilling, "Application of dynamic slicing in program debugging," *Proceedings of the 3rd International Workshop on Automatic Debugging* (*AADEBUG '97*), pp. 43–58. Linkping, Sweden, 1997.

[47] R.J. Lipton, "New directions in testing," *Proceedings of the DIMACS Workshop on Distributed Computing and Cryptography*, pp. 191–202. American Mathematical Society, Providence, RI, 1991.

[48] C. Liu, L. Fei, X. Yan, S.P. Midkiff, and J. Han, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[49] C. Liu, X. Yan, and J. Han, "Mining control flow abnormality for logic error isolation," *Proceedings of the 6th SIAM International Conference on Data Mining* (*SDM 2006*). Philadelphia, PA: Society for Industrial and Applied Mathematics, 2006.

[50] A.J. Offutt and E.J. Seaman, "Using symbolic execution to aid automatic test data generation," *Systems Integrity, Software Safety, and Process Security: Proceedings of the 5th Annual Conference on Computer Assurance* (*COMPASS '90*), pp. 12–21. Los Alamitos, CA: IEEE Computer Society, 1990.

[51] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. New York, NY: Wiley, 2008.

[52] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundation of Software Engineering* (*ESEC 2005/FSE-13*), pp. 263–272. New York, NY: ACM, 2005.

[53] T.H. Tse, T.Y. Chen, and Z.Q. Zhou, "Testing of large number multiplication functions in cryptographic systems," *Proceedings of the 1st Asia-Pacific Conference on Quality Software* (*APAQS 2000*), pp. 89–98. Los Alamitos, CA: IEEE Computer Society, 2000.

[54] T.H. Tse, F.C.M. Lau, W.K. Chan, P.C.K. Liu, and C.K.F. Luk, "Testing object-oriented industrial software without precise oracles or results," *Communications of the ACM*, vol. 50, no. 8, pp. 78–85, 2007.

[55] E.J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

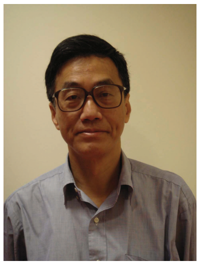
[56] S. Wolfram, *The Mathematica Book*. Wolfram Media, 2003.

[57] 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.

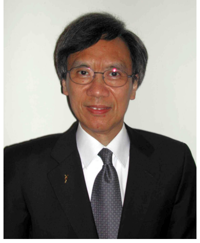
[58] A. Zeller, "Isolating cause-effect chains from computer programs," *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2002/FSE-10*), pp. 1–10. New York, NY: ACM, 2002.

[59] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

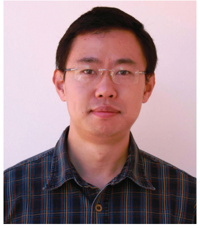
[60] Z.Q. Zhou, T.H. Tse, F.-C. Kuo, and T.Y. Chen, "Automated functional testing of Web search engines in the absence of an oracle," Technical Report TR-2007-06, Department of Computer Science, The University of Hong Kong, Hong Kong, 2007.



Tsong Yueh Chen received the BSc and MPhil degrees from The University of Hong Kong, the MSc degree and DIC from Imperial College London, and the PhD degree from The University of Melbourne. He is currently the chair professor of software engineering and the director of the Centre for Software Analysis and Testing at Swinburne University of Technology, Australia. His research interests include software testing, debugging, software maintenance, and software design. He is a member of the IEEE.



T.H. Tse received the PhD degree in information systems from the London School of Economics. He is a professor in computer science at The University of Hong Kong. He was a visiting fellow at the University of Oxford. His current research interest is in program testing, debugging, and analysis. He is the steering committee chair of QSIC and an editorial board member of the *Journal of Systems and Software*, *Software Testing, Verification and Reliability*, and *Software: Practice and Experience*. He is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and its Applications, and a fellow of the Hong Kong Institution of Engineers. He was decorated with an MBE by The Queen of the United Kingdom. He is a senior member of the IEEE.



Zhi Quan Zhou received the BSc degree in computer science from Peking University, China, and the PhD degree in software engineering from The University of Hong Kong. He is currently a lecturer in software engineering at the University of Wollongong, Australia. His research interests include software testing, debugging, software maintenance, symbolic analysis, and quality assessment of Web search engines.