Postprint of acticle in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '02*), ACM, New York, NY, pp. 191–195 (2002)

Semi-Proving: an Integrated Method Based on Global Symbolic Evaluation and Metamorphic Testing [*] [†]

T. Y. Chen
School of Information Technology,
Swinburne University of Technology,
Hawthorn 3122, Australia
tychen@swin.edu.au

T. H. Tse [‡]
Department of Computer Science,
The University of Hong Kong,
Pokfulam,
Hong Kong
thtse@cs.hku.hk

Zhiquan Zhou
Department of Computer Science,
The University of Hong Kong,
Pokfulam,
Hong Kong
zhiquan@uow.edu.au

ABSTRACT

We present a semi-proving method for verifying necessary conditions for program correctness. Our approach is based on the integration of global symbolic evaluation and metamorphic testing. It is relatively easier than conventional program proving, and helps to alleviate the problem that software testing cannot show the absence of faults.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Proving

General Terms

Verification

Keywords

Semi-proving, program proving, program testing, metamorphic testing, symbolic execution, global symbolic evaluation.

1. INTRODUCTION

Program proving and program testing are the two means of verifying the correctness of a program. The former uses a mathematical proof to show that the program in question is correct [12, 15]. It is, however, not popular in the industry because of the problems of automation and the complexity of proofs even for small programs. The latter, program testing, has remained the major means of establishing confidence in software correctness. Nevertheless, there are two fundamental problems in program testing, namely the reliable test set problem [13] and the oracle problem [18].

A *testing oracle* is a mechanism against which testers can check the output of a program and decide whether it is correct. The *oracle problem* refers to the fact that an oracle may not be available in many situations [18], such as during the computation of complex numerical analysis functions.

A common approach for testing numerical and scientific programs is to check whether they preserve some identity relations exhibited by the functions in question. Consider, for example, a program to compute the log function. Since we know that $\log x^2 = 2 \log x$, we should check whether the program results also demonstrate this property. This technique of verifying identity relations has been extensively used, for instance, in Cody and Waite [9].

Weyuker [18] defined a program to be "non-testable" if an oracle does not exist or is practically too difficult to determine. She investigated various alternative approaches to testing non-testable programs. In most cases, she proposed to make use of the theoretical properties of the target functions, including properties outlined in the last paragraph.

There is a closely related technique, known as *data diversity*, developed by Ammann and Knight [2]. The notion of data diversity is based on a very novel idea aimed at overcoming the problems associated with *N*-version programming. It has been developed from the perspective of fault tolerance rather than fault detection (and since then has only been advocated as a fault tolerance technique). As a consequence, properties used in data diversity are intrinsically limited to identity relations.

Since then, more research has been carried out in the area of automatic program testing without the need of a human oracle. The techniques of *program checkers*, for instance, were developed [1, 3]. Let p be a program purportedly computing function f, and x be an input case for p. A program checker is an algorithm that checks whether $p(x) = f(x)$ with a probability level specified by the user. In essence, this technique also utilizes the expected properties of the functions.

More recently, a *metamorphic testing* method was proposed by Chen et al. [4, 5]. It has been proposed as a property-based test case selection strategy. It is based on the intuition that even if no failure is revealed by a test case selected according to some strategies, it still has useful information. Thus, follow-up test cases should be further constructed from the original test cases with reference to some necessary conditions of the problem to be implemented.

[†] This research is supported in part by the Hong Kong Research Grants Council and the University of Hong Kong Committee on Research and Conference Grants.

[‡] Corresponding author.

Such necessary properties guiding the construction of follow-up test cases are known as *metamorphic relations*.

There are fundamental differences between metamorphic testing and the other methods outlined above, even though all of them propose to test programs against selected properties of implemented functions when testing oracles are not readily available:

(*a*) In metamorphic testing, the properties are not limited to identity relations. It has been applied, for instance, to verify the convergence of solutions of partial differential equations with respect to the refinement of grid points [6]. On the other hand, apart from a couple of examples on error bounds given by Weyuker, all the other techniques have only made use of identity relations.

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

(*c*) For program checkers, the main objective is to provide a probabilistic oracle for any given test case, so as to estimate whether the corresponding output is likely to be correct. The generation of additional test cases is only a by-product of the system. Metamorphic testing, however, does not aim at providing an alternative oracle for a given input. Instead, it postulates that even though we do not know whether a test case is successful in the absence of an oracle, its result may still carry very useful information: We can run a follow-up test case and compare the series of results against selected properties of the implemented functions. If the results do not exhibit the expected properties, the program must be at fault. In other words, metamorphic testing is a property-based test case selection strategy that can be used along with other test case selection strategies.

In Section 2, we shall present a semi-proving method that can be used either to prove that a program preserves selected necessary conditions of target functions, or to identify counterexamples if otherwise. Our method is based on the integration of metamorphic testing and global symbolic evaluation. Section 5 concludes the paper.

2. A FIRST EXAMPLE

2.1 Proving Necessary Conditions for Program Correctness

We shall introduce the semi-proving method through examples. The first example is the program "*GetMid*" shown in Figure 1. It accepts three real numbers $x1$, $x2$, and $x3$ as inputs and returns their median. It is adapted from [16], where it was used as a worst-case example to illustrate the constraint-based test case generation technique for mutation testing. We use this example to demonstrate how to prove that a program satisfies the expected metamorphic relations for *all* inputs. We can also guarantee that, if the program terminates but does not satisfy the expected relations for some input cases, these cases can *always* be identified and hence reveal an error in the program. Programs that do not terminate will be out of the scope of this paper.

```
1:  double GetMid (double x1, double x2, double x3) {
2:      double mid;
3:      mid = x3;
4:      if (x2 < x3)
5:          if (x1 < x2)
6:              mid = x2;
7:          else {
8:              if (x1 < x3)
9:                  mid = x1;
10:         }
11:     else
12:         if (x1 > x2)
13:             mid = x2;
14:         else if (x1 > x3)
15:             mid = x1;
16: return mid;
17: }
```

Figure 1: Program *GetMid*

Even for programs that terminate, it is not easy to prove their correctness. Most software engineers will resolve to test it by means of test cases. We propose to check the correctness of a program by proving selected metamorphic relations with respect to the function. Our first step is to identify some metamorphic relations. Let *Mid* be the function that we want the program to compute. We shall write the function as $Mid(x, y, z)$ and the corresponding program as $GetMid(X, Y, Z)$. When there is no confusion, we shall simply write them as $Mid(I)$ and $GetMid(I)$, respectively. An obvious property of the *Mid* function is that $Mid(\pi(I)) = Mid(I)$ for any input tuple I and any permutation $\pi(I)$ of I, such as $\pi(x, y, z) = (z, x, y)$. We shall verify whether the program *GetMid* also satisfies this property for all elements in the input domain. We note that all the permutations of the tuple I, together with the composition of permutations, form a *group* [14]. From group theory, any permutation of $I = (X, Y, Z)$ can be expressed as compositions of the transpositions $\tau_1(I) = (X, Z, Y)$ and $\tau_2(I) = (Y, X, Z)$. In other words, in order to prove that $GetMid(\pi(I)) = GetMid(I)$ for any input tuple $I = (X, Y, Z)$ and any permutation of I, we need only prove two properties, namely $GetMid(X, Z, Y) = GetMid(X, Y, Z)$ and $GetMid(Y, X, Z) = GetMid(X, Y, Z)$.

We execute *all* the possible paths in the program using the *global symbolic evaluation* technique [7, 8, 11], to produce three different symbolic outputs depending on different path conditions:

$$GetMid(X, Y, Z) = \begin{cases} X & \text{when condition } C1 \text{ holds} \\ Y & \text{when condition } C2 \text{ holds} \\ Z & \text{when condition } C3 \text{ holds,} \end{cases} \quad (1)$$

where condition $C1$ is $Y \le X < Z$ **or** $Z < X \le Y$,
condition $C2$ is $X < Y < Z$ **or** $Z \le Y < X$, and
condition $C3$ is $Y < Z \le X$ **or** $X \le Z \le Y$.

Let $\pi(X, Y, Z) = (X, Z, Y)$ be a permutation of (X, Y, Z). By global symbolic evaluation again, we have

$$GetMid(X, Z, Y) = \begin{cases} X & \text{when condition } C4 \text{ holds} \\ Z & \text{when condition } C5 \text{ holds} \\ Y & \text{when condition } C6 \text{ holds,} \end{cases} \quad (2)$$

where condition $C4$ is $Z \le X < Y$ **or** $Y < X \le Z$,
condition $C5$ is $X < Z < Y$ **or** $Y \le Z < X$, and
condition $C6$ is $Z < Y \le X$ **or** $X \le Y \le Z$.

We would like to prove that $GetMid(X,\ Z,\ Y) = GetMid(X,\ Y,\ Z)$ for any input $(X,\ Y,\ Z)$. According to equation (1), we need to prove this under the conditions $C1$, $C2$, and $C3$. When condition $C1$ holds, the output of $GetMid(X,\ Y,\ Z)$ has a value equal to X. We need only prove that the output of $GetMid(X,\ Z,\ Y)$ has the same value. We have three subcases:

(a) *Condition* $C4$ *is true:* In this situation, according to equation (2), the output of $GetMid(X,\ Z,\ Y)$ also has a value equal to X.

(b) *Condition* $C5$ *is true:* In this situation, we have ($Y \le X < Z$ or $Z < X \le Y$) and ($X < Z < Y$ or $Y \le Z < X$), which is a contradiction. Hence, this subcase can never occur.

(c) *Condition* $C6$ *is true:* In this situation, we have ($Y \le X < Z$ or $Z < X \le Y$) and ($Z < Y \le X$ or $X \le Y \le Z$), which can be simplified to ($X = Y < Z$) **or** ($Z < Y = X$). Hence, according to equation (2), the output of $GetMid(X,\ Z,\ Y)$ has a value equal to $Y = X$.

Thus, the outputs of $GetMid(X,\ Z,\ Y)$ and $GetMid(X,\ Y,\ Z)$ agree with each other when condition $C1$ holds. The cases when conditions $C2$ and $C3$ hold can be proved similarly. As a result, $GetMid(X, Z, Y) = GetMid(X, Y, Z)$ for any input (X, Y, Z).

Following the same procedure, we can also prove that $GetMid(Y, X, Z) = GetMid(X, Y, Z)$ for any input (X, Y, Z). According to group theory, therefore, we can conclude that $GetMid(\pi(I)) = GetMid(I)$ for any input tuple I and any permutation $\pi(I)$ of I. In other words, we have proved that the program is correct with respect to this metamorphic relation.

2.2 Detecting Program Faults

In the above example, we have demonstrated how to prove that a program satisfies a metamorphic relation. In this section, we shall create a fault in the program and demonstrate how the same method can be used to reveal the error. Let us remove statements 14 and 15 from the program *GetMid* in Figure 1, giving a faulty program $\overline{GetMid}$. This is known as a *missing path error*, generally considered "the most difficult type of error to detect by automated means" [16].

We follow the same semi-proving procedure as described earlier. Suppose we would like to verify the same metamorphic relation $\overline{GetMid}(\pi(I)) = \overline{GetMid}(I)$ for any input tuple I and any permutation $\pi(I)$ of I. By global symbolic evaluation, the output of $\overline{GetMid}$ is as follows:

$$\overline{GetMid}(X,\ Y,\ Z) = \begin{cases} X & \text{when condition } C7 \text{ holds} \\ Y & \text{when condition } C8 \text{ holds} \\ Z & \text{when condition } C9 \text{ holds,} \end{cases} \quad (3)$$

where condition $C7$ is $Y \le X < Z$,
condition $C8$ is $X < Y < Z$ **or** $Z \le Y < X$, and
condition $C9$ is $Y < Z \le X$ **or** ($Z \le Y$ **and** $X \le Y$).

First, let us verify that $\overline{GetMid}(X,\ Z,\ Y) = \overline{GetMid}(X, Y, Z)$. According to equation (3), we need to do this under the conditions $C7$, $C8$, and $C9$. Consider the case when condition $C7$ holds. The output of $\overline{GetMid}(X,\ Y,\ Z)$ has a value equal to X. By global symbolic evaluation,

$$\overline{GetMid}(X,\ Z,\ Y) = \begin{cases} X & \text{when condition } C10 \text{ holds} \\ Z & \text{when condition } C11 \text{ holds} \\ Y & \text{when condition } C12 \text{ holds,} \end{cases} \quad (4)$$

where condition $C10$ is $Z \le X < Y$,
condition $C11$ is $X < Z < Y$ **or** $Y \le Z < X$, and
condition $C12$ is $Z < Y \le X$ **or** ($Y \le Z$ **and** $X \le Z$).

```
/* Program Trap implements the trapezoidal rule to find the approximate
area under the curve f(x) between x = a and x = b. The computation uses
v intervals of size |b − a| / v. The variable "error" will be set to "true"
when n is less than 1. */
   float Trap(float (*f)(float), float a, float b, int v, bool & error) {
      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:          error = true;
         else {
3:          error = 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;
                  /* The denominator 2.0 will be modified to seed a fault. */
13:               yOld = yNew;
                  }
14:            area = area * h;
15:            if (a > b)
16:               area = −area;
               } }
17:      return area;
         }
```

Figure 2: Program $Trap$

According to equation (4), we have three subcases, depending on whether condition $C10$, $C11$, or $C12$ is true. We can skip the case where condition $C10$ holds, because the corresponding output is also X. When condition $C11$ holds, the combined constraint ($C7$ **and** $C11$) is a contradiction, and hence this subcase is impossible. When condition $C12$ holds, the combined constraint ($C7$ **and** $C12$) can be simplified to ($Y = X < Z$) **or** ($Y < X < Z$). When $Y < X < Z$, however, $\overline{GetMid}(X,\ Z,\ Y) \neq \overline{GetMid}(X,\ Y,\ Z)$ because $Y \neq X$. Thus, by identifying input cases such that $Y < X < Z$, we can prove that $\overline{GetMid}$ is faulty.

3. A SECOND EXAMPLE

In the previous example, the fault in the program is not within any loop. In this section, we shall illustrate how our method can be applied in situations where potential faults occur inside loops. For programs with loops, techniques for *loop analysis* or *loop generalization* [7, 8, 10, 17] are usually needed. As an example, consider a program $Trap$ adapted from [7] and shown in Figure 2. The program supposedly computes the approximate area under the curve $f(x)$ between $x = a$ and $x = b$. Through this example, we shall further illustrate how we can prove that a program satisfies expected necessary conditions.

First, we need to identify a metamorphic relation. Suppose $G(x) = F(x) + C$, where C is a positive constant. From elementary calculus, we know that $Trap(G,\ A,\ B,\ V,\ ERROR) = Trap(F, A, B, V, ERROR) + C \times |B - A|$ when $V \ge 1$, where the symbol "ERROR" is a Boolean constant with a value of "true" or "false". We apply the loop analysis technique [7] in global symbolic evaluation and obtain the following result. There are all together six classes of execution paths for any input $(F,\ A,\ B,\ V,\ ERROR)$. The paths, path

conditions, and corresponding symbolic outputs are as follows:

(1) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17).
Path condition: $V = 1$ **and** $A > B$.
area: $(F(A)/2.0 + F(B)/2.0) \times (A - B)$.
error: false.

(2) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17).
Path condition: $V = 1$ **and** $A < B$.
area: $(F(A)/2.0 + F(B)/2.0) \times (B - A)$.
error: false.

(3) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, $(9,\ 10,\ 11,\ 12,\ 13)^+$, 14, 15, 16, 17), where $(9,\ 10,\ 11,\ 12,\ 13)^+$ denotes two or more iterations of the subpath enclosed by the parentheses. The same notation will be used throughout the rest of this section.
Path condition: $V > 1$ **and** $A > B$ **and** $k_e = V$, where $k_e = \min\{k \in \mathbf{N} : k > 1$ **and** $-B + k \times (B - A)/V + A \leq 0\}$.
area: $(F(A)/2.0 + F(B)/2.0 + \sum_{i=1}^{V-1} F(A - A \times i/V + B \times i/V)) \times (A/V - B/V)$.
error: false.

(4) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, $(9,\ 10,\ 11,\ 12,\ 13)^+$, 14, 15, 17).
Path condition: $V > 1$ **and** $B > A$ **and** $k_e = V$, where $k_e = \min\{k \in \mathbf{N} : k > 1$ **and** $-B + k \times (B - A)/V + A \geq 0\}$.
area: $(F(A)/2.0 + F(B)/2.0 + \sum_{i=1}^{V-1} F(A - A \times i/V + B \times i/V)) \times (B/V - A/V)$.
error: false.

(5) ***Path:*** (0, 1, 3, 4, 5, 17).
Path condition: $V \geq 1$ **and** $A = B$.
area: 0.
error: false.

(6) ***Path:*** (0, 1, 2, 17).
Path condition: $V < 1$.
area: undefined.
error: true.

Let us consider an input case $(G,\ A,\ B,\ V,\ ERROR)$, where $G(x) = F(x) + C$ and $C > 0$. We do a global symbolic evaluation of the program again for this input. We note that the change of the first parameter from F to G does not affect the selection of execution paths when the program is being run. In other words, both the paths to be executed and the path conditions are the same for $Trap(F,\ A,\ B,\ V,\ ERROR)$ and $Trap(G,\ A,\ B,\ V,\ ERROR)$. The only difference is the symbolic value of the variable *area* after execution. In the following results, we shall not list the path conditions, since they are identical to the corresponding ones above.

(1) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17).
area: $((F(A) + C)/2.0 + (F(B) + C)/2.0) \times (A - B)$.
error: false.

(2) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17).
area: $((F(A) + C)/2.0 + (F(B) + C)/2.0) \times (B - A)$.
error: false.

(3) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, $(9,\ 10,\ 11,\ 12,\ 13)^+$, 14, 15, 16, 17).
area: $((F(A) + C)/2.0 + (F(B) + C)/2.0 + \sum_{i=1}^{V-1}(F(A - A \times i/V + B \times i/V) + C)) \times (A/V - B/V)$.
error: false.

(4) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, $(9,\ 10,\ 11,\ 12,\ 13)^+$, 14, 15, 17).
area: $((F(A) + C)/2.0 + (F(B) + C)/2.0 + \sum_{i=1}^{V-1}(F(A - A \times i/V + B \times i/V) + C)) \times (B/V - A/V)$.
error: false.

(5) ***Path:*** (0, 1, 3, 4, 5, 17).
area: 0.
error: false.

(6) ***Path:*** (0, 1, 2, 17).
area: undefined.
error: true.

Now we need to verify whether $Trap(G,\ A,\ B,\ V,\ ERROR) - Trap(F,\ A,\ B,\ V,\ ERROR) = C \times |B - A|$ when $V \geq 1$ for every possible path. Since the global symbolic evaluation of the program produces six classes of paths, there are all together $6 \times 6 = 36$ combinations of path conditions to be considered. As 30 of them are contradictions, however, we need only consider the remaining six of them. For example, for the first path with the condition ($V = 1$ **and** $A > B$), we calculate $Trap(G,\ A,\ B,\ V,\ ERROR) - Trap(F,\ A,\ B,\ V,\ ERROR) = ((F(A) + C)/2.0 + (F(B) + C)/2.0) \times (A - B) - (F(A)/2.0 + F(B)/2.0) \times (A - B) = C \times (A - B) = C \times |A - B|$, which satisfies the metamorphic relation. Following this procedure, we can prove that the outputs of all the remaining paths satisfy the metamorphic relation. [1] Hence, the program $Trap$ satisfies the metamorphic relation for *any* input data.

Let us now seed a fault into statement 12. Suppose we change the denominator "2.0" into "2.01", simulating a typo, and see how it can be revealed by the method. We shall denote the faulty program by $\overline{Trap}$. We do not need to complete the entire global symbolic evaluation before verifying the expected metamorphic relation. On the contrary, verification can start immediately after any selected path has been executed. (In this sense, semi-proving is also a symbolic testing method.) For example, the symbolic execution of $\overline{Trap}(F, A, B, V, ERROR)$ on the first path will produce

(1) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17).
Path condition: $V = 1$ **and** $A > B$.
area: $(F(A)/2.01 + F(B)/2.01) \times (A - B)$.
error: false.

Then, we perform the second symbolic execution for the same path and condition, this time using $\overline{Trap}(G,\ A,\ B,\ V,\ ERROR)$. We obtain the following result:

(1) ***Path:*** (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17).
area: $((F(A) + C)/2.01 + (F(B) + C)/2.01) \times (A - B)$.
error: false.

Consider the first path with the path condition ($V = 1$ **and** $A > B$). With respect to the above pair of results, we have $\overline{Trap}(G, A, B, V, ERROR) - \overline{Trap}(F, A, B, V, ERROR) = ((F(A) + C)/2.01 + (F(B) + C)/2.01) \times (A - B) - (F(A)/2.01 + F(B)/2.01) \times (A - B) = (2/2.01) \times C \times (A - B) = (2/2.01) \times C \times |A - B| \neq C \times |A - B|$. Thus, a fault has been detected. The fault can also be detected when other paths are being executed and reviewed.

We recognize that faults such as that in statement 12 may also be uncovered by program testing. However, even though we can select test cases to cover a given path, there is no guarantee in conventional program testing that all faults in that path will be detected. Furthermore, when testers execute the program using real number test cases, minor discrepancies may easily be overlooked because small differences are expected in floating-point arithmetic. This situation can be illustrated by the following example: Let $f(x) = \sin^3 x \times \cos x$, $v = 100000$, $a = -PI$, and $b = 2 \times PI$, where PI is set to 3.14159265. The area computed by the original (correct) program is 0.000206829. For a faulty program with the denominator in statement 12 replaced by "2.001", the output is 0.000206765, which is only different from the correct output by 0.000000064. This small difference may generally be anticipated in floating-point arithmetic and hence ignored by mistake. On the other hand, using the semi-proving method with symbolic input, the error is guaranteed to be uncovered.

[1] Actually, we need only prove this for the first five of the remaining paths because, for the last case, $V < 1$ and is therefore not applicable.

4. SUMMARY OF SEMI-PROVING PROCEDURE

Only two examples have been used in the previous subsections to illustrate how an integrated method based on metamorphic testing and global symbolic evaluation can be used (*a*) to verify expected necessary properties and (*b*) to identify failure-causing inputs if such properties are not satisfied. We observe, because of the page limit of the paper, not all steps have been included in the examples, and not all steps have been performed in full. For readers' benefit, therefore, a summary of the whole semi-proving procedure is listed below:

/* Procedure V verifies whether program p satisfies the expected metamorphic relation R_p. V will end quietly if p satisfies R_p throughout the input domain. Otherwise, V will identify all the failure-causing inputs in the form of mathematical constraints. Without loss of generality, we assume that only two executions of the program will be required for verifying the selected metamorphic relations. */

1: procedure V(p, R_p) {
2: generate first symbolic input case I;
3: do global symbolic evaluation $p(I)$ to produce all possible outputs $O_1, O_2, \ldots, O_n$ under path conditions $C_1, C_2, \ldots, C_n$;
4: generate second symbolic input case (or cases) I' according to R_p;
5: do global symbolic evaluation $p(I')$ to produce all possible outputs $O'_1, O'_2, \ldots, O'_m$ under path conditions $C'_1, C'_2, \ldots, C'_m$;
6: for each path condition C_i, $i = 1, 2, \ldots, n$ {
7: for each path condition C'_j, $j = 1, 2, \ldots, m$ {
8: if (C_i **and** C'_j) is not a contradiction {
9: if $(I, I', O_i, O'_j) \notin R_p$ under the condition (C_i **and** C'_j)
10: print "Fault detected for inputs I and I' under metamorphic constraint (C_i **and** C'_j)";
11: } } } }

5. CONCLUSION

We have presented a semi-proving method that verifies expected necessary conditions for program correctness. This method is an integration of symbolic evaluation and metamorphic testing techniques. Our method involves both structural (white-box) information when performing global symbolic evaluation, and functional (black-box) information of the problem domain when identifying metamorphic relations. By combining black- and white-box information, subtle errors in white-box testing such as missing path errors can be better tackled. From the perspective of testing, our integrated approach helps to alleviate the problem that software testing does not affirm the absence of faults. From the perspective of program proving, target properties can be verified using appropriate sets of inputs identified through metamorphic relations. We have also outlined a procedure for proving selected necessary conditions of program correctness and identifying failure-causing inputs, if any.

Although property testing has long been recognized in software testing, it has been used only as a stand-alone technique. This paper has shown the potential of integrating property testing with other verification techniques.

6. ACKNOWLEDGEMENTS

We would like to thank Phyllis Frankl of the Polytechnic University, Brooklyn for her information and discussions on symbolic evaluation techniques, and the anonymous reviewers for pointing out some relevant references that we have overlooked.

7. REFERENCES

[1] L. M. Adleman, M.-D. Huang, and K. Kompella. Efficient checkers for number-theoretic computations. *Information and Computation*, 121 (1): 93–102, 1995.

[2] P. E. Ammann and J. C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Transactions on Computers*, 37 (4): 418–425, 1988.

[3] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42 (1): 269–291, 1995.

[4] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering* (*SE '98*), pages 191–197. ACTA Press, Calgary, Canada, 1998.

[5] 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.

[6] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC '02*). IEEE Computer Society, Los Alamitos, CA, 2002.

[7] L. A. Clarke and D. J. Richardson. Symbolic evaluation methods: implementations and applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–102. North-Holland, Amsterdam, 1981.

[8] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5: 15–35, 1985.

[9] W. J. Cody, Jr. and W. Waite. *Software Manual for the Elementary Functions*. Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[10] D. D. Dunlop and V. R. Basili. Generalizing specifications for uniformly implemented loops. *ACM Transactions on Programming Languages and Systems*, 7 (1): 137–158, 1985.

[11] P. G. Frankl. Partial symbolic evaluation of path expressions. Computer Science Technical Report PUCS-105-90. Department of Electrical Engineering and Computer Science, Polytechnic University, Brooklyn, New York, NY, 1989.

[12] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8 (3): 331–353, 1976.

[13] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2 (3): 208–215, 1976.

[14] W. Ledermann and A. J. Weir. *Introduction to Group Theory*. Longman, Harlow, Essex, UK, 1996.

[15] A. Mili. *An Introduction to Formal Program Verification*. Van Nostrand Reinhold, New York, NY, 1985.

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

[17] R. C. Waters. A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, SE-5 (3): 237–247, 1979.

[18] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4): 465–470, 1982.