Utilizing Hoare Logic to Strengthen Testing for Error Detection in Programs*

Shaoying Liu

Hosei University, Japan
sliu@hosei.ac.jp

Abstract

Hoare logic (also known as Floyd-Hoare logic) can be used to formally verify the correctness of programs while testing provides a practical way to detect errors in programs. Unfortunately, the former is rarely applied in practice and the later is difficult to detect all existing errors. In this paper, we propose a novel technique that makes good use of Hoare logic to strengthen testing. The essential idea is first to use specification-based testing to discover all traversed program paths and then to use Hoare logic to prove their correctness. During the proof process, all errors on the paths can be detected. A case study is conducted to show its feasibility; an example taken from the case study is used to illustrate how the proposed method is applied; and discussion on the potential challenges to the method is presented.

1 Introduction

Formal verification (or proof) based on Hoare logic [1] provides a possibility to establish the correctness for programs, but due to the difficulty in deriving appropriate invariants for iterations in general, as well as the difficulties in managing *side effect* and invocations of subroutines (methods, functions, or procedures) in programming languages (e.g., Java, C#), formal proof for realistic programs is rarely used in practice.

Specification-based testing (SBT) is a practical technique for detecting program errors. A strong point of SBT superior to formal correctness verification is that it is much easier to be performed automatically if formal specifications are adopted [2, 3], but a weak point is that existing errors on a program path may still not be uncovered even if it has been traversed using a test case. We believe that the strong point of SBT should be utilized to realize full automation for error detection efficiency, but its weak point should be overcome by making good use of relevant part of Hoare logic.

In this paper, we put forward a novel approach to testing programs by combining a specific SBT we have developed before with the Hoare logic-based formal correctness verification. This new approach is known as *testing-based formal verification* (TBFV). The essential idea is first to generate a test case from each functional scenario, derived from the formal specification using pre- and post-conditions, to run the program, and then repeatedly apply the axiom for assignment in Hoare logic to formally verify the correctness of the path that is traversed by using the test case. As described in Section 2, any pre-post style formal specification can be automatically transformed into an equivalent disjunction of functional scenarios and each scenario defines an independent function of the corresponding program in terms of the relation between input and output. A test case can be generated from each functional scenario and can be used to run the program to find a traversed path, which is a sequence of conditions or statements, but the correctness of the path with respect to the pre-condition and the functional

*This work is supported by NII Collaborative Program, SCAT research foundation, and Hosei University.

scenario is unlikely to be established by means of testing. This deficiency can be eliminated by repeatedly applying the axiom for assignment in Hoare logic. The superiority of our approach to both SBT and formal verification is that it can verify the correctness of all traversed paths and can be performed automatically because the derivation of invariants from iterations is no longer needed.

Our focus in this paper is on the explanation of the new idea in combining specification-based testing with Hoare logic. Therefore, we deliberately choose small examples to explain the principle, which is expected to facilitate the reader in understanding the essential idea. The feasibility of applying the new technique to deal with a realistic program system has been demonstrated in our case study.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to both the functional scenario-based testing (FSBT) and the axiom for assignment in Hoare logic. Section 3 describes the essential idea of our TBFV approach. In Section 4, we give an example to illustrate the TBFV approach systematically. Section 5 elaborates on how method invocation is dealt with in TBFV. Section 6 gives a brief overview of the related work. Finally, in Section 7, we conclude the paper and point out future research direction and topics.

2 Introduction to FSBT and Hoare Logic

This section briefly introduces the relevant parts of FSBT and Hoare logic to pave the way for the discussion of our TBFV approach.

2.1 FSBT

FSBT is a specific specification-based testing approach that takes both the pre-condition and post-condition into account in test case generation [3]. Applying the principle of "divide and conquer", the approach treats a specification as a disjunction of *functional scenarios* (FS), and to generate test sets and analyze test results based on the functional scenarios. A functional scenario in a pre-post style specification is a logical expression that tells clearly what condition is used to constrain the output when the input satisfies some condition.

Specifically, let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the specification of an operation S, where S_{iv} is the set of all input variables whose values are not changed by the operation, S_{ov} is the set of all output variables whose values are produced or updated by the operation, and S_{pre} and S_{post} are the pre- and post-conditions of S, respectively. The characteristic of this style specification is that the post-condition S_{post} is used to describe the relation between initial states and final states. We assume that in the post-condition, a decorated variable, such as $\tilde{}x$, is used to denote the initial value of external (or state) variable x before the operation and the variable itself, i.e., x, is used to represent the final value of x after the operation. Thus, $\tilde{}x \in S_{iv}$ and $x \in S_{ov}$. Of course, S_{iv} also contains all other input variables declared as input parameters and S_{ov} includes all other output variables declared as output parameters.

A practical strategy for generating test cases to exercise the behaviors expected of all functional scenarios derived from the specification is established based on the concept of functional scenario. To precisely describe this strategy, we first need to introduce functional scenario.

Definition 1. *Let* $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \cdots \vee (C_n \wedge D_n)$*, where each* C_i $(i \in \{1, ..., n\})$ *is a predicate called "*guard condition*" that contains no output variable in* S_{ov}*;* D_i *a "*defining condition*" that contains at least one output variable in* S_{ov} *but no guard condition. Then, a* functional scenario f_s *of* S *is a conjunction* $\tilde{}S_{pre} \wedge C_i \wedge D_i$*, and the expression*

$(\tilde{}S_{pre} \wedge C_1 \wedge D_1) \vee (\tilde{}S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\tilde{}S_{pre} \wedge C_n \wedge D_n)$ *is called a* functional scenario form *(*FSF*) of* S*.*

The decorated pre-condition $\tilde{}S_{pre} = S_{pre}(\tilde{}\sigma/\sigma)$ denotes the predicate resulting from substituting the initial state $\tilde{}\sigma$ for the final state σ in pre-condition S_{pre}. We treat a conjunction $\tilde{}S_{pre} \wedge C_i \wedge D_i$ as a scenario because it defines an independent behavior: when $\tilde{}S_{pre} \wedge C_i$ is satisfied by the initial state (or intuitively by the input variables), the final state (or the output variables) is defined by the defining condition D_i. The conjunction $\tilde{}S_{pre} \wedge C_i$ is known as the *test condition* of the scenario $\tilde{}S_{pre} \wedge C_i \wedge D_i$, which serves as the basis for test case generation from this scenario.

To support automatic test case generation from functional scenarios, the vital first step is to obtain an FSF from a given specification. A systematic transformation procedure, algorithm, and software tool support for deriving an FSF from a pre-post style specification have been developed in our previous work [4]. Generating test cases based on a specification using the functional scenario-based test case generation method is realized by generating them from its all functional scenarios. The production of test cases from a functional scenario is done by generating them from its test condition, which can be divided further into test case generations from every disjunctive clause of the test condition. In the previous work [3], a set of criteria for generating test cases are defined in detail. To effectively apply FSBT, the FSF of the specification must satisfy the *well-formed* condition defined below.

Definition 2. *Let the FSF of specification* S *be* $(\tilde{}S_{pre} \wedge C_1 \wedge D_1) \vee (\tilde{}S_{pre} \wedge C_2 \wedge D_2) \vee \cdots \vee (\tilde{}S_{pre} \wedge C_n \wedge D_n)$*. If* S *satisfies the condition* $(\forall_{i,j \in \{1,\ldots,n\}} \cdot (i \neq j \Rightarrow (C_i \wedge C_j \Leftrightarrow false))) \wedge (\tilde{}S_{pre} \Rightarrow (C_1 \vee C_2 \vee \cdots \vee C_n \Leftrightarrow true))$*,* S *is said to be well-formed.*

The well-formedness of specification S ensures that each functional scenario defines an independent function and the guard conditions completely cover the restricted domain (a subdomain of the operation in which all of the values satisfy the pre-condition). Thus, for any input satisfying the pre-condition, S is guaranteed to define an output satisfying the defining condition of only one functional scenario.

Under the assumption that S is well-formed, we can focus on test case generation from a single functional scenario, say $\tilde{}S_{pre} \wedge C_i \wedge D_i$, at a time using our approach. The test case is then used to run the program, which will enable one program path to be executed. Let us take operation $ChildFareDiscount$, a process of the *IC card system for JR commute train service* used in our case study that is briefly explained in Section 4, as an example. The functionality of the process is specified using the SOFL specification language [5] below, which is similar to VDM-SL for operation specifications.

process $ChildFareDiscount(a : int, n\_f : int)\ a\_f : int$
pre $a > 0$ *and* $n\_f > 1$
post $(a > 12 \Rightarrow a\_f = n\_f)$
and
$(a \leq 12 \Rightarrow a\_f = n\_f - n\_f * 0.5)$
end_process

The specification states that the input a (standing for *age*) must be greater than 0 and n_f ($normal\_fare$) must be greater than 1. when a is greater than 12, the output a_f ($actual\_fare$) will be the same as n_f ; otherwise, a_f will be 50% discount on n_f.

According to the algorithm reported in our previous work [4], three functional scenarios can be derived from this formal specification:

(1) $a > 0$ *and* $n\_f > 1$ *and* $a > 12$ *and* $a\_f = n\_f$
(2) $a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$ *and* $a\_f = n\_f - n\_f * 0.5$

Table 1: A test example

test case:	$a = 5$, $n\_f = 2$
test condition:	$a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$
functional scenario:	$a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$ *and* $a\_f = n\_f - n\_f * 0.5$

(3) $a \leq 0$ *or* $n\_f \leq 1$ *and anything*

where *anything* means that anything can happen when the pre-condition is violated.

Assume the formal specification is refined into the following program (a Java-like method):

int ChildFareDiscount(*int a*, *int n_f*) {
(1) *If* ($a > 0$ && $n\_f > 1$){
(2) *if* ($a > 12$)
(3) $a\_f := n\_f$;
(4) *else* $a\_f := n\_f * *2 - n\_f - n\_f * 0.5$;
(5) *return* a_f;}
(6) *else System.out.println*("*the precondition is violated.*");
}

where the symbol := is used as the assignment operator in order to distinguish from the equality symbol = used in the specification. It is evident that we can derive the following paths: [(1)(2)(3)(5)], [(1)(2)′(4)(5)], and [(1)′(6)]. In the path [(1)(2)′(4)(5)], (2)′ means the negation of the condition $a > 12$ (i.e., $a \leq 12$), and the similar interpretation applies to (1)′ in path [(1)′(6)]. We also deliberately insert a defect in the assignment $a\_f = n\_f * *2 - n\_f - n\_f * 0.5$ (the correct one should be $a\_f = n\_f - n\_f * 0.5$), where $n\_f * *2$ means n_f to the power 2 (i.e., n_f^2).

The weakness of the testing approach is that it can only find the presence of errors but cannot find their absence. For example, we generate a test case, $\{(a, 5), (n\_f, 2)\}$, from the test condition $a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$ of functional scenario (2), as illustrated in Table 1. Executing the program with this test case, the path [(1)(2)′(4)(5)] will be traversed. The result of the execution is $a\_f = 2 * *2 - 2 - 2 * 0.5 = 1$. This result does not indicate the existence of error because when the test condition $a > 0$ *and* $n\_f > 1$ *and* $a \leq 12$ is satisfied by the test case, the defining condition $a\_f = n\_f - n\_f * 0.5$ is also satisfied by the output $a\_f = 1$ (because $1 = 2 - 2 * 0.5 <\Rightarrow true$), which proves that *in this case*, the program correctly implements the functional scenario. But obviously the path contains an error.

One solution to this problem is to perform a formal verification based on Hoare logic to check whether the traversed path is correct with respect to the functional scenario. The correctness verification is expected to be fully automatic in order to allow us to integrate this technique into the functional scenario-based testing approach. To understand this point further, we need to briefly introduce the axiom for assignment in Hoare logic.

2.2 Hoare logic

Hoare logic is established based on predicate logic and provides a set of axioms to define the semantics of programming languages. For each program construct, such as sequence, selection, or iteration, an axiom for defining its semantics is defined. These axioms can be used to reason about the correctness of programs written in a programming language. For the sake of space and the relevance of the axioms to this paper, we only introduce the axiom for assignment in this section.

Let $x := E$ be an assignment: assigning the result of evaluating expression E to variable x. The axiom for assignment is

$$\overline{\{Q(E/x)\}\ x := E\ \{Q\}}\ .$$

It states that the assignment $x := E$ is correct with respect to the given post-assertion Q and the derived pre-assertion $Q(E/x)$, a predicate resulting from substituting E for all of the free occurrences of x in Q. The post-assertion Q presents a condition that must be satisfied by variable x after the execution of the assignment (the assignment can be treated as an operation that updates variable x). To make post-assertion Q true after the execution, expression E must satisfy Q before the execution, that is, $Q(E/x)$ is true, because x represents E after the execution.

3 Principle of TBFV

TBFV proposed in this paper provides a specific technique for verifying the correctness of traversed program paths identified using FSBT. The principle underlying the technique includes the following three points:

- Using FSBT to generate adequate test cases to identify all of the *representative paths* in the program under testing; each path is traversed by using at least one test case. A representative path is formed by treating an iteration as an *if-then-else* construct to ensure that the body of the iteration is executed at least once and the iteration terminates, and by treating all of the other constructs as same as their original form.

- Let $\tilde{}S_{pre} \wedge C_i \wedge D_i$ $(i = 1, ..., n)$ denote a functional scenario and test case t be generated from the test condition $\tilde{}S_{pre} \wedge C_i$. Let $p = [sc_1, sc_2, ..., sc_m]$ be a program path in which each sc_j $(j = 1, ..., m)$ is called a *program segment*, which is a decision (i.e., a predicate), an assignment, a "return" statement, or a printing statement. Assume path p is traversed by using test case t. To verify the correctness of p with respect to the functional scenario, we form a *path triple*

  $\{\tilde{}S_{pre}\}\ p\ \{C_i \wedge D_i\}$.

  The path triple is similar in structure to Hoare triple, but is specialized to a single path rather than the whole program. It means that if the pre-condition $\tilde{}S_{pre}$ of the program is true before path p is executed, the post-condition $C_i \wedge D_i$ of path p will be true on its termination.

- Repeatedly applying the axiom for assignment or the axiom we provide below for other relevant statements, we can derive a pre-assertion, denoted by p_{pre}, to form the following expression:

  $\{\tilde{}S_{pre}(\tilde{}x/x)\}\ \{p_{pre}(\tilde{}x/x)\}\ p\ \{C_i \wedge D_i(\tilde{}x/x)\}$.

  where $\tilde{}S_{pre}(\tilde{}x/x)$, $p_{pre}(\tilde{}x/x)$ and $C_i \wedge D_i(\tilde{}x/x)$ are a predicate resulting from substituting every decorated input variable $\tilde{}x$ for the corresponding input variable x in the corresponding predicate, respectively. These substitutions are necessary to avoid confusion between the input variables and the internally updated variables (which may share the same name as the input variables).

  Finally, if the implication $\tilde{}S_{pre}(\tilde{}x/x) \Rightarrow p_{pre}(\tilde{}x/x)$ can be proved, it means that no error exists on the path; otherwise, it indicates the existence of some error on the path.

The axiom for the other relevant statements or decisions is given as follows:

$\overline{\{Q\}S\{Q\}}$,

where S is one of the three kinds of program segments: *decision*, "*return*" *statement*, and *printing statement*. The axiom describes that the pre-condition and post-condition for any of the three kinds of program segments are the same because none of them changes states. We call this *axiom for non-change segment*.

It is worth mentioning that since the application of the axioms for assignment and for non-change segment involves only syntactical manipulation, deriving the pre-assertion $p_{pre}(\tilde{}x/x)$ can be automatically carried out, but formally proving the implication $\tilde{}S_{pre}(\tilde{}x/x) \Rightarrow p_{pre}(\tilde{}x/x)$, which we simply write as $\tilde{}S_{pre} \Rightarrow p_{pre}$ below in this paper, may not be done automatically, even with the help of a theorem prover such as PVS, depending on the complexity of $\tilde{}S_{pre}$ and p_{pre}. If achieving a full automation is regarded as the highest priority, as taken in our approach, the formal proof of this implication can be "replaced" by a test. That is, we first generate sample values for variables in $\tilde{}S_{pre}$ and p_{pre}, and then evaluate both of them to see whether p_{pre} is false when $\tilde{}S_{pre}$ is true. If this is true, it tells that the path under examination contains an error. Since the testing technique is already available in the literature [3, 6], we do not repeat the detail in this paper for the sake of space. Our experience suggests that in many realistic circumstances, testing can be both practical and beneficial. However, if the correctness assurance is regarded as the highest priority, a formal proof of the implication must be performed.

4 Example

We have conducted a case study to apply our TBFV approach to test and verify a simplified version of the *IC card system for JR commute train service* in Tokyo. Our experience shows that the approach is feasible and can be effective in general but also faces some challenges or limitations that need to be addressed in the future research. The system we used is designed to offer the following functional services: (1) Controlling access to and exit from a railway station, (2) Buying tickets using the IC card, (3) Recharging the card by cash or through a bank account, and (4) Buying a railway pass for a certain period (e.g., for one month or three months). Due to the limit of space, we cannot present all of the details, but take one of the internal operations used in the system, which is *ChildFareDiscount* mentioned above, as an example to illustrate how TBFV is applied to rigorously test the corresponding program. The program contains three paths, it is necessary to formally verify all of the three paths. Since the process of the verification is the same for all the paths, we only use the path $[(1)(2)'(4)(5)]$ that is traversed by using the test case $\{(a, 5), (n\_f, 2)\}$ as an example for explanation.

Firstly, we form the path triple:

$\{\tilde{}a > 0 \ and \ \tilde{}n\_f > 1\}$
$[a > 0 \ \&\& \ n\_f > 1$
$a \leq 12,$
$a\_f := n\_f * *2 - n\_f - n\_f * 0.5,$
$return \ a\_f \ ]$
$\{\tilde{}a \leq 12 \ and \ a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

where $\tilde{}a > 0 \ and \ \tilde{}n\_f > 1$ is the result of substituting $\tilde{}a$ and $\tilde{}n\_f$ for input variables a and n_f, respectively, in the pre-condition of the program, and $\tilde{}a \leq 12 \ and \ a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5$ is the result of completing the similar substitution in the post-condition.

Secondly, we repeatedly apply the axiom for assignment or the one for non-change segment to this path triple, starting from the post-condition. As a result, we form the following path, known as *asserted path*, with derived internal assertions between two program segments:

$\{\tilde{}a > 0\ and\ \tilde{}n\_f > 1\}$
$\{\tilde{}a \leq 12\ and$
$\tilde{}n\_f * *2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a > 0\ \&\&\ n\_f > 1$
$\{\tilde{}a \leq 12\ and$
$n\_f * *2 - n\_f - n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

$a \leq 12$
$\{\tilde{}a \leq 12\ and$
$n\_f * *2 - n\_f - n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a\_f := n\_f * *2 - n\_f - n\_f * 0.5$
$\{\tilde{}a \leq 12\ and\ a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$return\ a\_f$
$\{\tilde{}a \leq 12\ and\ a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

where the assertion $\tilde{}a \leq 12\ and\ \tilde{}n\_f * *2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$, the second from the top of the sequence, is the result of substituting $\tilde{}a$ for a and $\tilde{}n\_f$ for n_f in the derived assertion $\{\tilde{}a \leq 12\ and\ n\_f * *2 - n\_f - n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$. As explained previously, this is necessary in order to keep consistency of the input variables a and n_f in the original pre-condition (appearing as $\tilde{}a$ and $\tilde{}n\_f$) and the derived pre-assertion.

Thirdly, we need to judge the validity of the implication $\tilde{}a > 0\ and\ \tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12\ and\ \tilde{}n\_f * *2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$. Using the test case $\{(\tilde{}a, 5), (\tilde{}n\_f, 8)\}$, we can easily prove that the implication is false (the evaluation detail is omitted due to space limit).

From this example, we can see that sometimes testing can be even more efficient than formal proof in judging the validity of the implication when an error exists on the path, but if the path contains no error, testing will be almost impossible to give a firm conclusion in general. In that case, an engineering judgement must be made about the validity. The good point about testing is that complete automation can be realized, which is extremely helpful for industry.

5 Dealing with method invocation

If a method invocation is used as a statement, it may change the current state of the program. Therefore, the traversed path within the invoked method will have to be taken into account in deriving the pre-assertion of the program under testing.

Let us change the program *ChildFareDiscount* and organize the implementation into a class called *FareDiscount* below.

```
        class FareDiscount {
        int tem;  //instance variable

         int ChildFareDiscount1(int a, int n_f) {
(1)       Discount(n_f);
(2)       if (a > 0 && n_f > 1){
(3)        if (a > 12)
(4)           a_f := n_f;
(5)        else a_f := n_f * *2 − n_f − tem;
(6)       return a_f; }
(7)      else System.out.println("the precondition is violated.");
         }

        void Discount(int x){
        int r;
(1.1)   r := x * 0.5;
(1.2)   tem := r; }
        }
```

When running the method $ChildFareDiscount1$ in which the method $Discount(n\_f)$ is invoked, we obtain three paths: [(1)(2)(3)(4)(6)], [(1)(2)(3)′(5)(6)], and [(1)(2)′(7)], where segment (1) is a subpath $[(1.1)(1.2)](n\_f/x)$, denoting the path resulting from substituting actual parameter n_f for formal parameter x in the subpath [(1.1)(1.2)]. Thus, [(1)(2)(3)′(5)(6)] for example, actually means the path after inserting the traversed path in $Discount$ into the traversed path in $ChildFareDiscount1$, which is simply represented by [(1.1)(1.2)(2)(3)′(5)(6)]. Selecting the same test case $\{(a, 5), (n\_f, 2)\}$ as before to run the program, we make the path [(1.1)(1.2)(2)(3)′(5)(6)] traversed. We then form the asserted path as follows:

$\{\tilde{}a > 0$ *and* $\tilde{}n\_f > 1\}$
$\{\tilde{}a \leq 12$ *and*
$\tilde{}n\_f * *2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$r := n\_f * 0.5$
$\{\tilde{}a \leq 12$
$n\_f * *2 - n\_f - r = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$tem := r$
$\{\tilde{}a \leq 12$ *and*
$n\_f * *2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a > 0$ && $n\_f > 1$
$\{\tilde{}a \leq 12$ *and*
$n\_f * *2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a \leq 12$
$\{\tilde{}a \leq 12$ *and*
$n\_f * *2 - n\_f - tem = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
$a\_f := n\_f * *2 - n\_f - tem$
$\{\tilde{}a \leq 12$ *and* $a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$
return a_f
$\{\tilde{}a \leq 12$ *and* $a\_f = \tilde{}n\_f - \tilde{}n\_f * 0.5\}$

where the subpath $[r := n\_f * 0.5, tem := r]$ is the result of substituting actual parameter n_f used in the method invocation $Discount(n\_f)$ for formal parameter x used in the method definition in the original subpath $[r := x * 0.5, tem := r]$. Similarly, we can easily use testing to

prove that the implication $\tilde{}a > 0$ *and* $\tilde{}n\_f > 1 \Rightarrow \tilde{}a \leq 12$ *and* $\tilde{}n\_f * *2 - \tilde{}n\_f - \tilde{}n\_f * 0.5 = \tilde{}n\_f - \tilde{}n\_f * 0.5$ is false, indicating that an error is found on the path.

6 Related Work

Research on integration of Hoare logic and testing seems to mainly concentrate on using pre- and post-assertions in Hoare triple for test case generation and test result analysis, but none of them takes the same approach as our TBFV to solve the same problem in specification-based testing.

One of the earliest efforts is Meyer's view of Design By Contract (DBC) implemented in the programming language Eiffel [7, 8]. Eiffel's success in checking pre- and post-conditions and encouraging the DBC discipline in programming partly contributed to the development of the similar work for other languages such as the Sunit testing system for Smalltalk [9]. Cheon and Leavens describe an approach to unit testing that uses a formal specification language's runtime assertion checker to decide whether methods are working correctly with respect to a formal specification using pre- and post-conditions, and have implemented this idea using the Java Modeling Language (JML) and the JUnit testing framework [10]. Gray and Mycroft describe another approach to testing Java programs using Hoare-style specifications [11]. They show how logical test specifications with a more relaxed post-condition than existing restricted Hoare-style post-condition can be embedded within Java and how the resulting test specification language can be compiled into Java for executable validation of the program. There are many other similar results in the literature, but we have to omit them due to the space limit.

7 Conclusion and future research

We presented a new approach, known as testing-based formal verification (TBFV), for error detection in programs by integrating specification-based testing and Hoare logic. The principle underlying TBFV is first to use the functional scenario-based testing (FSBT) to achieve a (representative) path coverage in the program under testing, and then to apply the Hoare logic-based approach to formally verify the correctness of every traversed path. Since both techniques for FSBT and for path correctness verification can be automatically performed, the TBFV approach has an advantage over formal correctness verification based on Hoare logic in dealing with realistic program systems. It also has a potential advantage in reducing the number of necessary test cases over the existing specification-based testing.

While focusing on the presentation of the essential idea of the TBFV approach and an example from the case study to show its feasibility and potential effectiveness in this paper, a controlled experiment needs to be conducted to systematically assess the effectiveness and to compare with the related testing and formal verification approaches. Further research is also needed to address the tool support issue.

References

[1] C. A. R. Hoare and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[2] S. Khurshid and D. Marinov. TestEra: Specification-based Testing of Java Programs using SAT. *Automated Software Engineering*, 11(4), 2004.

[3] S. Liu and S. Nakajima. A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications. In *4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010)*, pages 147–155, Singapore, June 9-11 2010. IEEE CS Press.

[4] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima. Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation. In *9th International Conference on Software Methodologies, Tools and Techniques (SoMet 2010)*, page to appear, Yokohama city, Japan, Sept. 29- Oct. 1 2010. IOS International Publisher.

[5] S. Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[6] S. Liu and S. Nakajima. A "Vibration" method for Automatically Generating Test Cases Based on Formal Specifications. In *18th Asia-Pacific Software Engineering Conference (APSEC 2011)*, pages 73–80, HCM City, Vietnam, Dec. 5-8 2011. IEEE CS Press.

[7] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, Oct. 1992.

[8] B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series, 1991.

[9] M. C. Castellon, J. G. Molina, E. Pimentel, and I. Repiso. Design by contract in smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, Dec. 1996.

[10] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unity Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP 2002*, pages 231–255, Spain, 2002. LNCS 2374, Springer-Verlag.

[11] K. E. Gray and A. Mycroft. Logical Testing: Hoare-style Specification Meets Executable Validation. In *Proceedings of 2009 Fundamental Approaches to Software Engineering (FASE 2009)*, pages 186–200, York, UK, March 2009. LNCS 5503, Springer-Verlag.