Testing-Based Formal Verification on Software Programs: A Review

Ai Liu
Hiroshima University
liuai@hiroshima-u.ac.jp

Shaoyin Liu
Hiroshima University
sliu@hiroshima-u.ac.jp

Abstract

Testing-based formal verification has been proposed to automatically verify whether a software program satisfies the requirements written as a formal specification by exploring program paths. There are two tools that can be used to verify path correctness: Hoare logic or symbolic execution. In this document, we will discuss their merits and demerits, as well as the future research directions.

1. Introduction

Software testing, one of the most frequently used quality assurance methods, is focused on sampling the executions, according to some coverage criteria, and comparing the actual behavior with the expected behavior. Nevertheless, a typical weakness of testing is that it cannot generally tell the absence of bugs in programs. To find more bugs, more test data need to be generated and more program executions, which will inevitably increasing the cost. An idea for improvement is to ensure the correctness of the executed path for a given test case with formal methods and then all of the test cases, resulting in the same executed path, will satisfy the expected behavior.

Formal methods, based on some mathematical theories, are a collection of notations and techniques, including specification and verification [1]. Formal specification techniques introduce a precise and unambiguous description of the software behavior. Given such a specification, it is possible to use formal verification techniques to demonstrate that a software design is correct with respect to the specification. However, most of software verification methods do not guarantee the correctness of actual code, but rather allow one to verify some abstract model of it. Hence, due to possible differences between the actual software system and its abstract model, the correctness of the proof may not carry over. To ensure the correctness of the code, deductive verification, including symbolic execution and Hoare logic, has been demonstrated mainly for simple programs which contain no loop sentences and no side effects.

Hoare logic is established based on predicate logic and provides a set of axioms to define the semantics of programming language [2]. 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. However, the axiom for iteration is hard to be directly used since the derivation of loop invariants may not be easy.

Symbolic execution systematically explores the possible paths of a program's execution by executing the program with symbolic inputs instead of concrete values and then allows the tester to explore different execution paths and detect potential bugs and vulnerabilities [3]. Symbolic execution is often used in combination with other techniques such as model checking and theorem proving to verify the correctness of software systems. There are severe challenges of symbolic execution such as path explosion and constraint solving. Nevertheless, it is an active area of research, and new techniques and tools are constantly being developed to improve its efficiency and effectiveness.

A natural idea is to combine testing and formal methods (specification and verification) to make the best use of their merits while reducing the cost. As Whittaker noted "Without a specification testers are likely to find only the most obvious bugs" in 2000, specification-based testing (SBT) has been advocated for a long time [4, 5]. However, due to the weakness of testing to tell the absence of bugs in programs, specification-based testing also need high test cost in order to improve software reliability. For the purpose to improve specification-based testing with formal verification techniques, a method known as testing-based formal verification (TBFV), characterized by the integration of specification-based testing with Hoare logic to ensure the correctness of program paths which contain no loop sentences and no side effects, has been ex-

plored and its initial idea is proposed in [6]. The idea is further developed to combine with program inspection later [7]. A collection of prototype tools to illustrate how to support the TBFV method is presented in [8]. The original framework of TBFV is extended in [9] to deal with operations, which may involve with complex data structure and side effects, in software packages, such as Vector, ArrayList and LinkedList packages in Java language. Before that extension, testing-based formal verification with symbolic execution is alternatively proposed as an improved method by replacing the application of Hoare logic with symbolic execution in [10]. Branch sequence coverage (BSC) for TBFV-SE is considered in [11] and a fault localization approach is proposed to further pinpoint the problematic positions in the incorrect program paths given by TBFV-SE in [12]. Moreover, the idea is also applied to the verification of neural networks, resulting in TBFV-INN, a new framework for verifying and improving neural networks [13].

As previously mentioned, since TBFV is firstly proposed, it has been constantly improved on two roads: one is based on Hoare logic and the other is based on symbolic execution. In this document, we will enjoy different scenery and discuss which treasure we intend to dig on the two roads in the future.

The rest is organized as follows. Section 2 introduces the framework of testing-based formal verification. Section 3 describes the framework of test-based formal verification with symbolic execution. Section 4 discusses the merits and demerits of TBFV and TBFV-SE.

2. The Framwork of TBFV

In formal methods, the first step to implement an operation is usually to write a formal specification to describe its requirements. The assumption for TBFV is that the operation specification S can be represented as a functional scenario form $\bigvee_{i=1}^{n}(T_i \wedge D_i)$ where T_i is called a test condition and D_i is called a defining condition. A test condition is a constraint only on the input variables, while a defining condition contains at least one output variable. The functional scenario $T_i \wedge D_i$ describes a single specific functional behavior: when T_i is true, the output of the operation is defined using D_i. Given an operation specification $S = \bigvee_{i=1}^{n}(T_i \wedge D_i)$ with a program P, the TBFV method is proposed to tackle the problem whether both input and output variables satisfy D_i after P is executed if input variables satisfy the corresponding T_i.

The framework of TBFV is illustrated in Figure 1 and explained as follows.

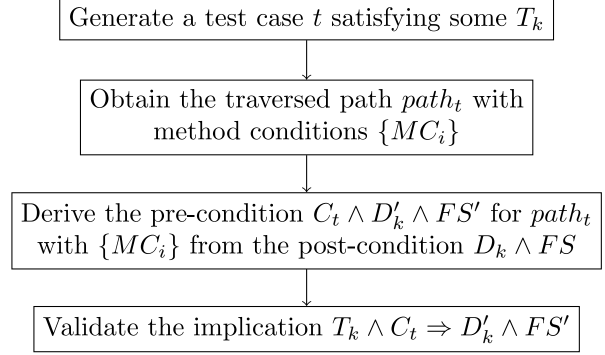

Figure 1. The framework of TBFV

- The first step is to randomly generate a test case t satisfying some test condition T_k. Only one test condition will be satisfied due to the well-formedness of the specification. Note that when the program involves with a method invocation of a complex component (such as Vector, ArrayList and LinkedList in JAVA), the test case t needs to contain the initial component state.
- The second step is to execute program P with test case t and obtain a traversed program path $path_t$, which is a sequence of guard conditions, assignment statements and method invocations. Each method has already been equipped with a Hoare tripe which may contain multiple activated guard conditions. The activated guard condition MC_i of the i-th invoked method should also be recorded.
- As the test case t must contain the initial component state, correspondingly, the post-condition should be not only the defining condition D_k, but the conjunction of the defining condition and the final component state $D_k \wedge FS$ (each component has a final state representation FS). Assuming such post-condition, we can backwards derive the pre-condition $C_t \wedge D'_k \wedge FS'$ of $path_t$ with method conditions $\{MC_i\}$ where C_t is the conjunction of guard conditions in $path_t$ after replacing all intermediate variables with expressions of input variables and $D'_k \wedge FS'$ is obtained by replacing all output variables with expressions of input variables in the derivation process.
- The final step is to find whether $T_k \wedge C_t$ implies $D'_k \wedge FS'$ with the aid of theorem provers. If it is a tautology, $path_t$ with method conditions $\{MC_i\}$ is correct with respect to the specification S; otherwise, there must be errors in $path_t$.

If the path correctness is ensured, all test cases validating $T_k \wedge C_t$ will satisfy the specification. Then, we

can choose a test case t validating $T_k \wedge \neg C_t$ to continue the above steps.

Example 1. *Consider a jump function f whose domain is the set of integers $\mathbb{Z}$ and output is*

$$y = f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Then, the corresponding FSF S is

$$(x \in \mathbb{Z} \wedge x \geq 0 \wedge y = 1) \vee (x \in \mathbb{Z} \wedge x < 0 \wedge y = 0)$$

Consider the following program:

```
int jump(int x){
    int y = 0;
    if (x >= 0)
        y = (x + 2)/(x + 1);
    return y;
}
```

where variable x is the input and variable y is the output. If the given input is $x = 1$, the corresponding testing condition is $x \in \mathbb{Z} \wedge x \geq 0$ and defining condition is $y = 1$. Then we can derive the pre-condition for the corresponding path as follows

$$\{x \geq 0 \wedge 1 = (x + 2)/(x + 1)\}$$
$$\mathtt{y = 0}$$
$$\{x \geq 0 \wedge 1 = (x + 2)/(x + 1)\}$$
$$\mathtt{x >= 0}$$
$$\{1 = (x + 2)/(x + 1)\}$$
$$\mathtt{y = (x + 2)/(x + 1)}$$
$$\{y = 1\}$$

The next step is to consider the implication

$$x \in \mathbb{Z} \wedge x \geq 0 \wedge x \geq 0 \Rightarrow 1 = (x + 2)/(x + 1)$$

which is true for $x > 0$ due to the division operation of integers but false when $x = 0$. If we only use testing, we may select many test cases in which $x \neq 0$ and produce the test results as expected. In this case, the contradiction can not be revealed. However, using TBFV, we can get the same path and implication once a test case $x \geq 0$ is chosen, and find the contradiction.

3. The Framework of TBFV-SE

Testing-based formal verification with symbolic execution (TBFV-SE) has already been a method to automatically verify the correctness of all the representative program paths given the formal specifications and derived a fault localization approach. In TBFV-SE, the correctness of each path is also converted into a theorem obtained during a dynamic symbolic execution, rather than the derivation by Hoare logic. The framework of TBFV-SE is illustrated in Figure 2 and explained as follows.

- Naturally, the first step is also to generate a test case t, but there is no need to fix a test condition. The initial component state should be included in the test case if some component method is invoked.
- After executing program P with test case t, a traversed program path can be obtained. Replacing test case t with symbol values and applying symbolic execution for that path, all the output variables, path conditions and component states are finally related to the symbol values. We use C_t and $state_t$ to denote path-condition, the integration of all path conditions and a symbolic state including all output variables and all component states, respectively.
- Although the test case t only validates a testing condition, there may exist other test cases, validating different testing conditions, which result in the same traversed path. If the implication $C_t \wedge T_i \Rightarrow state_t \wedge D_i$ is tautologous, it means that the test cases validating testing condition T_i and resulting in this traversed path satisfy the specification. There, the theorem we intend to prove is $\forall i \; C_t \wedge T_i \Rightarrow state_t \wedge D_i$, which ensures the path correctness.
- Branch sequence coverage (BSC) is proposed to automatically partition the domain by dynamically gathering all the necessary symbolic paths since each symbolic path is selected to represent a sub-domain. This partition is depended on a dynamic light analysis of code structure without judging if the branch conditions are serialized or nested. Unlike conventional partition testing selecting classical values from classes to test, BSC focuses on automatically selecting a real symbolic path for representing each sub-domain according to the real structure of codes. There is an algorithm to automatically find test cases satisfying BSC in [11].
- A fault localization algorithm is proposed in [12] to provide a rigorous way to automatically analyze the correctness of statements in a program based on the theorems of the traversed paths and help locate the faults in a small set of statements by examining very small percentage of the code.

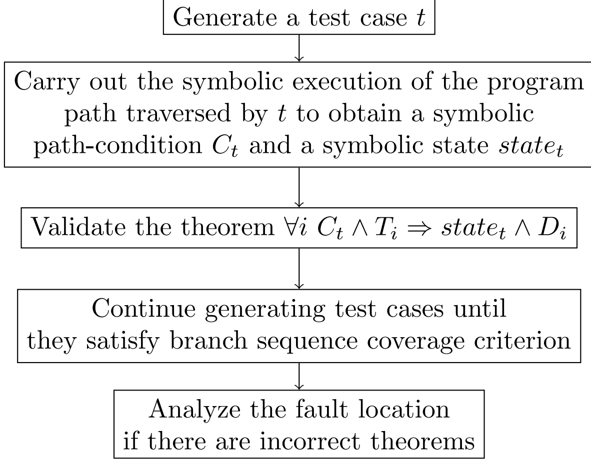

Figure 2. The framework of TBFV-SE

Example 2. *Reconsidering Example 1 with TBFV-SE, the derivation of path-condition and the record of states for the traversed path by the input x with numeric value* 1 *and symbolic value* X *is displayed in Table 1. The last line of second column of Table 1 shows both the symbolic state and symbolic path-condition. Then, the induced theorem is*

$$(\mathtt{x} \in \mathbb{Z} \wedge \mathtt{x} \geq 0 \wedge \mathtt{x} \geq 0 \Rightarrow y = 1 \wedge y = \frac{\mathtt{X}+2}{\mathtt{X}+1}) \wedge$$
$$(\mathtt{x} \in \mathbb{Z} \wedge \mathtt{x} < 0 \wedge \mathtt{x} \geq 0 \Rightarrow y = 0 \wedge y = \frac{\mathtt{X}+2}{\mathtt{X}+1})$$

where the first sub-formula is false while the second sub-formula is true. We just use this simple example to show how symbolic execution help derive theorems in TBFV-SE.

Table 1. Single path of symbolic execution

program code	symbolic execution
int jump(**int** x)	input: $x = 1$ symbolic value of x : X
int y = 0	state: $\{y = 0\}$
if (x >= 0)	path-condition: $\mathtt{X} \geq 0$
y = (x + 2)/(x + 1)	state: $\{y = \frac{\mathtt{X}+2}{\mathtt{X}+1}\}$
return y	$\{y = \frac{\mathtt{X}+2}{\mathtt{X}+1}\}, \mathtt{X} \geq 0$

4. Merits and Demerits

As for our goal to verify whether a program satisfies its corresponding specification, TBFV and TBFV-SE can both automatically generate test cases, obtain traversed paths and induce theorems. Next, external theorem provers will be used to validate the induced theorems. Further, TBFV-SE has been enriched with two algorithms to find representative program paths and analyze the correctness of statements. We mainly compare the common procedures of TBFV and TBFV-SE.

Given a test case t, TBFV and TBFV-SE will obtain the same program path $path_t$. The difference is the way to derive the theorems, as depicted in Example 1 and 2. In TBFV, we need to preselect a functional scenario $T_i \wedge D_i$ and the derived theorem can only represent whether the traversed path satisfies that functional scenario. In TBFV-SE, the derived theorem can be used to judge whether the traversed path satisfies all functional scenarios. Given the test cases inducing a traversed path, if some of them validate a testing condition T_i and the rest validate another testing condition T_j, we just need one derivation process of the theorem in TBFV-SE while we should need two derivation processes of the theorem in TBFV. In that case, TBFV-SE should be more efficient than TBFV. Nevertheless, TBFV will have an advantage than TBFV-SE in cases including many intermediate variables and irrelevant variables, as depicted in Example 3.

Example 3. *Consider the following program which implements a piecewise function,*

```
int piece(float x){
    float y1 = 1 − x;
    float y2 = 2x;
    float y3 = 1/x;
    if (x < −4)
        y = y1;
    else if (−4 ≤ x ≤ 0)
        y = y2;
    else
        y = y3;
    return y;
}
```

For any input value, the traversed path must include the first three assignment statements. Assume the input is 0 *and the corresponding functional scenario is* $T \wedge D$ *where* D *only includes the variable* y*, the derivation by Hoare logic is below.*

$$\{-4 \leq x \leq 0 \wedge D(2x/y)\}$$
```
y1 = 1 − x
```
$$\{-4 \leq x \leq 0 \wedge D(2x/y)\}$$
```
y2 = 2x
```
$$\{-4 \leq x \leq 0 \wedge D(y2/y)\}$$

$$
\begin{aligned}
&\texttt{y3} = \texttt{1/x} \\
&\{-4 \le x \le 0 \wedge D(y2/y)\} \\
&-4 \le \texttt{x} \le 0 \\
&\{D(y2/y)\} \\
&\texttt{y} = \texttt{y2} \\
&\{D\}
\end{aligned}
$$

Then, the theorem to be proved in TBFV is

$$T \wedge -4 \le x \le 0 \Rightarrow D(2x/y)$$

The corresponding symbolic execution is shown in Table 2 where we omit the skipped statements. Then, the theorem to be proved in TBFV-SE is

$$
\begin{aligned}
&T \wedge \texttt{X} \ge 4 \wedge -4 \le \texttt{X} \le 0 \\
\Rightarrow &D \wedge y1 = 1 - \texttt{X} \wedge y2 = 2\texttt{X} \wedge y3 = \frac{1}{\texttt{X}} \wedge y = 2\texttt{X}
\end{aligned}
$$

Table 2. Symbolic execution for $x = 0$

program code	symbolic execution
int piece(**float** x)	input: $x = 0$ symbolic value of x : $\texttt{X}$
float $\texttt{y1} = 1 - \texttt{x}$	state:$\{y1 = 1 - \texttt{X}\}$
float $\texttt{y2} = 2\texttt{x}$	state:$\{y1 = 1 - \texttt{X}, y2 = 2\texttt{X}\}$
float $\texttt{y3} = 1/\texttt{x}$	state:$\{y1 = 1 - \texttt{X}, y2 = 2\texttt{X}$, $y3 = \frac{1}{\texttt{X}}\}$
if $\texttt{x} < -4$	path-condition $\texttt{X} \ge 4$
else if $-4 \le \texttt{x} \le 0$	path-condition $\texttt{X} \ge 4 \wedge$ $-4 \le \texttt{X} \le 0$
$\texttt{y} = \texttt{y2}$	state:$\{y1 = 1 - \texttt{X}, y2 = 2\texttt{X}$, $y3 = \frac{1}{\texttt{X}}, y = 2\texttt{X}\}$
return y	$\{y1 = 1 - \texttt{X}, y2 = 2\texttt{X}$, $y3 = \frac{1}{\texttt{X}}, y = 2\texttt{X}\}$, $\texttt{X} \ge 4 \wedge -4 \le \texttt{X} \le 0$

Obviously, the theorem obtained by TBFV-SE is more complex than the theorem obtained by TBFV, Actually, we must simplify the theorem obtained by TBFV-SE as

$$T \wedge \texttt{X} \ge 4 \wedge -4 \le \texttt{X} \le 0 \Rightarrow D \wedge y = 2\texttt{X}$$

which can be proved by theorem provers, since the formulas $y1 = 1 - \texttt{X}$, $y2 = 2\texttt{X}$ and $y3 = \frac{1}{\texttt{X}}$ are undecidable equations. Moreover, $D \wedge y = 2\texttt{X}$ can be further simplified as $D(2\texttt{X}/y)$. Therefore, the theorem obtained by TBFV is more concise and there is no need to record intermediate variables (e.g. $y2$) and irrelevant variables (e.g. $y1$ and $y3$) in TBFV.

5. Conclusion and Future Work

TBFV and TBFV-SE are both currently able to reduce test cost by ensuring path correctness, while each of them has its advantages and disadvantages. TBFV-SE includes a fault localization algorithm based on the traversed paths obtained by test cases satisfying branch sequence coverage criterion. In the future, we intend to locate faults in a single path which is proved to have errors by TBFV.

References

[1] Peled, D. A. "Software Reliability Methods", Springer, 2001.

[2] Hoare, C. A. R. and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica* 2, pp. 335-355, 1973.

[3] Cadar, C. and Sen, K. "Symbolic Execution for Software Testing: Three Decades Later", ACM, 2013.

[4] Whittaker, J. A. "What is software testing? Why is it so hard? Practice Tutorial", *IEEE Softw.* 17(1), pp. 70-79, 2000.

[5] Chen, Y. and Liu, S. "An Approach to Detecting Domain Errors Using Formal Specification-Based Testing", *Proceedings of APSEC 2004:* 276-283.

[6] Liu, S. "Utilizing Hoare Logic to Strengthen Testing for Error Detection in Program",*Proceedings of Turing-100 2012*, pp. 229-238.

[7] Liu, S. and Nakajima, S. "Combining Specification-Based Testing, Correctness Proof, and Inspection for Program Verification in Practice", *Proceedings of SOFL+MSVL 2013*, pp. 3-16.

[8] Liu, S. "A Tool Supported Testing Method for Reducing Cost and Improving Quality", *Proceedings of QRS 2016*, pp. 448-455.

[9] Liu, A. and Liu, S. "Enhancing the Capability of Testing-Based Formal Verification by Handling Operations in Software Packages", *IEEE Trans. Software Eng.* 49(1), pp. 304-324, 2023.

[10] Wang, R. and Liu, S. "TBFV-SE: Testing-Based Formal Verification with Symbolic Execution", *Proceedings of QRS 2018*, pp. 59-66.

[11] Wang, R. and Liu, S. “Branch Sequence Coverage Criterion for Testing-Based Formal Verification with Symbolic Execution”, *Proceedings of QRS-C 2019*, pp. 205-212.

[12] Wang, R. et al. “A Fault Localization Approach Derived from Testing-Based Formal Verification”, *Proceedings of ICECCS 2022*, pp. 165-170.

[13] Liu, H. et al. “Verifying and Improving Neural Networks Using Testing-Based Formal Verification”, *Proceedings of SOFL+MSVL 2022*, pp. 126-141.