Position Statement: Can Testing Prove Software Has No Bug?

Hong Zhu
Department of Computing and Communication Technologies, Oxford Brookes University
Oxford OX33 1HX, UK, e-mail: hzhu@brookes.ac.uk

"*Program testing can be used to show the presence of bugs, but never their absence*", Dijkstra alleged in 1972 [1]. However, software engineers have never stopped testing in practice. Instead, more emphasis has been put on software testing in modern software development methodologies, such as the so-called *test-driven* approach in agile methodologies [2]. Over many years of practice in software engineering, it is widely recognized that confidence in software systems can be gained from systematic testing. The question is whether there is a theoretical foundation to this claim.

Addressing this problem, in 1975, Goodenough and Gerhart made a significant breakthrough by proposing the notion of test adequacy criteria [3]. Since then, a large number of adequacy criteria have proposed and investigated C.f. [4]. Several theories have also been advanced in attempt to prove that testing can guarantee software correctness. These theories fall into two categories.

- *Statistical theories*. Consider the case when test cases are selected at random. The basic idea of this approach is: if a software system passes certain number of random tests, the reliability of the software system can be asserted with certain confidence according to the mathematical theory of probability and statistics.
- *Fault elimination theories*. This is based on the assumption that there are only a limited number of ways that a software system can contain faults. When the software passes a test case successfully, it can be regarded as eliminating certain possible faults in the system. After passing a large number of well selected test cases, most possible faults of the software can be eliminated. Fault-based software testing methods (such as mutation testing and perturbation testing) and error-based testing methods (such as category partitioning testing) have been advanced.

However, none of these two approaches can lead to a definite answer to the foundation problem of software test. Thus, we have proposed an *inductive inference theory* [5]. It regards software testing as an inductive inference process. Indeed, testing is induction because a tester observes a software system's correctness on a finite number of test cases and then tries to conclude that the system is correct on all inputs. However, in practice, such inference is done implicitly, even omitted completely.

There are a number of different computational inductive inference models that have been studied in the literature. One of the most well-known is *identification in the limit*. Applying this model to software testing led to results relating testing to software correctness. Here is a brief summary of the main results.

Let M be an inductive inference device, and $a = a_1, a_2, \ldots, a_n, \ldots$ be an infinite sequence of instances of a given rule f. Let $f_n = M\ \{(a_1, a_2, \ldots, a_n)\}$. If there is a natural number K such that for all $n, m \geq K$, $f_m = f_n$, then we say that M *converges* to f_K on a and that M *behaviorally identifies* f *correctly in the limit* by M. If the $f_K = f$, we say that M *explanatorily identifies* f *correctly in the limit*. A set P of rules is *behaviorally (or explanatorily) learnable* by M, if for all $f \in P$, f is behaviorally (or explanatorily) learnable by M.

For example, the set of one-variable polynomials is an explanatorily learnable set of functions. More examples of learnable rule sets can be found in [6].

The notions of software testing can be interpreted in the terminology of inductive inference as follows. A program under test is interpreted as a rule to learn. A test case is interpreted as an instance of the rule. A test set is then a set of instances. Consider P to be a set of functions on a domain D of input values such that both the program p under test and its specification s are included in P. With this interpretation, in [5], we have proved the following result.

Proposition 1. Let P be a behaviourally learnable set of functions on a domain D. Let $|\ .\ |$ be a complexity measure of the elements in D such that for any natural number N the subset $\{x \in D \mid |x| \leq N\}$ is finite. Then, for all functions $p, s \in P$, there exists a natural number N such that $p \equiv s$, if and only if $p(x) = s(x)$ for all $x \in D$ where $|x| \leq N$.

This theorem states that the correctness of a software system can be validated by testing on a finite number of test cases provided that the program and specification are in a learnable set of functions. Moreover, such testing can be performed without writing down a formal specification. This lays a foundation of the current practice of software testing where formal specifications are not available. This theorem implies that what current testing practice lacks is an analysis of the "complexity" of the program to determine a learnable set within which the program and the specification vary.

References

[1] E. W. Dijkstra, Notes on structured programming. In *Structured Programming,* by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press. 1972.

[2] Beck, K. Test-Driven Development, Addison Wesley, 2003.

[3] J. B. Goodenough, AND S. L. Gerhart, Toward a theory of test data selection. *IEEE Trans. Softw. Eng. SE-3,* June 1975.

[4] Zhu, H., Hall, P. and May, J., Software unit test coverage and adequacy, ACM Computing Survey, 29(4), Dec. 1997, pp366~427.

[5] Zhu, H., A formal interpretation of software testing as inductive inference, Journal of Software Testing, Verification and Reliability 6, July 1996, pp3~31.

[6] Case, J. & Smith, C., (1983) 'Comparison of identification criteria for machine inductive inference', *Theoretical Computer Science*, 25(2), 193~220.