Introduction to the Formal Treatment of Testing

John S. Gourlay
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio, USA

Abstract. Some of the major definitions of a mathematical framework for reasoning about testing are explained and used in the statement of a broad theorem that limits the power of specification independent, or structural, testing.

Introduction

Testing is an indispensable part of program development and, despite continuing work on program verification, it will surely continue to be necessary for many years. At the same time, theoretical computer science has been able to say very little to help testing practice or even to explain the obvious value of testing. While respecting and preserving program verification as an ideal, theory must be able to discuss degrees of program validation short of this ideal. It is the purpose of this paper to introduce an effort to fill this gap in theory.

The framework discussed here evolved as a synthesis of earlier work by Goodenough and Gerhart [2], Howden [4], and Geller [1]. It is elaborated and applied extensively in another paper by Gourlay [3]. Here we introduce some of the major definitions of the framework and indicate without proof the type of results that can be obtained within it.

The Mathematical Framework

We must begin a theoretical discussion of testing with an intuitive definition of the nature of the testing process. An idealized scenario will serve this purpose. First, a customer somehow specifies what is needed in the way of software. Second, a programmer takes the specification and attempts to write a program to meet it. Third, the programmer or another person takes both the program and the specification, and, based on these, selects a test to be performed on the program. Finally, the test (which may involve running the program several times on various data) is performed and the results are compared with the specification. Failure of the test indicates that the program is wrong and needs to be rewritten. Success of the test is taken to mean that the program is ready for use.

In practice, a specification is never initially complete enough to allow a programmer to walk away with it and write a program in isolation. In fact, programmers expect to return frequently to the customer to clarify ambiguities. For testing theory, however, the fact that the specification is composed in parallel with the program is not important, because

the end result is the same, namely that a specification is chosen and a program is written that purports to satisfy it.

Similarly, testing often goes on in parallel with programming. This also presents no conflict with the idealization, because the testing of components of the final program can be looked upon as the testing of individual programs with respect to their own separate specifications.

One important simplification of the real world does exist in this idealization, however. A typical test could comprise several runs of the program on various data. The assumption that the testing can be completed implies that each test run of the program can be completed in a reasonable amount of time. We cannot assume that every test in the world terminates. What we must assume is that no tester will deliberately set himself an infinitely long list of tests, and that part of every specification is (perhaps implicitly) a performance criterion that will allow the tester to decide whether or not a given test has run too long and has therefore failed.

The ability to complete the testing also implies that the success or failure of a test can be decided. We will accept this, observing that (at least in the obvious case of a test being a run of the program on a particular input) it should be easier to establish the correctness of a test than of the program as a whole.

It is our purpose to concentrate on the selection of tests and on the interpretation of the results of tests. For this reason we will not worry about where programs and specifications come from We will assume that we are simply given one of each and that the specification is correct in the sense that there is no other arbiter of the correctness of the program. The program is correct if and only if it satisfies the specification.

It is evident, then, that the theory of testing must be able to deal with the relationships between three sets of abstract objects: programs, specifications, and tests. The set of all programs will be denoted by P, the set of all specifications by S, and the set of all tests by T. p and q will denote programs, elements of P, r and s will denote specifications, elements of S, and t and u will denote tests, elements of T. Upper case letters will denote subsets of P, S, and T, so, for example, $p \in P \subseteq \mathsf{P}$.

The correctness of a program p with respect to a specification s will be denoted by $p \,\mathrm{corr}\, s$. Clearly, $\mathrm{corr} \subseteq \mathsf{P} \times \mathsf{S}$. We will also have use for another predicate, $\mathrm{ok} \subseteq \mathsf{P} \times \mathsf{S} \times \mathsf{T}$. $\mathrm{ok}(p, s, t)$ will be written as $p \,\mathrm{ok}_t\, s$ and will mean that test t performed on program p is judged successful by specification s.

In order for such a collection of sets and predicates to be a legitimate model of testing we need to know that if a program is correct with respect to a given specification, then no tests of that program will ever fail with respect to the specification. In other words, we need to know that $p \,\mathrm{corr}\, s \Rightarrow p \,\mathrm{ok}_t\, s$. Thus, we have the following.

Definition 1:

A *testing system* is a collection $(\mathsf{P}, \mathsf{S}, \mathsf{T}, \mathrm{corr}, \mathrm{ok})$ where P, S, and T are arbitrary sets, $\mathrm{corr} \subseteq \mathsf{P} \times \mathsf{S}$, $\mathrm{ok} \subseteq \mathsf{P} \times \mathsf{S} \times \mathsf{T}$, and $\forall p \forall s \forall t (p \,\mathrm{corr}\, s \Rightarrow p \,\mathrm{ok}_t\, s)$. ∎

Probably the simplest kind of testing system to imagine is one in which a test $t \in \mathsf{T}$ is a datum from the input domain of the programs in P. In this case, the natural interpretation of $p \,\mathrm{ok}_t\, s$ would be to run p on t and check to see that the result is satisfactory according to s. It is important to note, however, that in realistic testing systems, tests are generally

much more complex objects than individual data. Statement testing, for example, seeks to run the program on a diverse enough collection of data so that every statement of the program is executed at least once. This goal provides a rule under which certain sets of test data are considered adequate and others are not. To perform statement testing, we must (somehow) construct one of these adequate test test data sets, and then run the program on every datum in it. We then check to see that the program satisfies the specification on each of the test runs. In a testing system capable of representing statement testing, then, a $t \in \mathsf{T}$ must be a set of sets of input data. The interpretation of ok becomes correspondingly complex, but the details are beyond the scope of this short introduction. For the purpose of this paper, T will remain an uninterpreted set.

From here on, we will work implicitly within the testing system $(\mathsf{P}, \mathsf{S}, \mathsf{T}, \mathrm{corr}, \mathrm{ok})$. We are ready, now, to consider the process of choosing tests. In the idealization of the programming process, the tester chooses a test based on the program and specification he is given.

Definition 2:

A *test method* is a function $M : \mathsf{P} \times \mathsf{S} \rightarrow \mathsf{T}$. In other words, M takes a program and a specification and generates a test. ∎

Definition 3:

$p \mathrm{ok}_M s$ will abbreviate $p \mathrm{ok}_{M(p,s)} s$. In other words, if we let $t = M(p, s)$ then $p \mathrm{ok}_M s \Leftrightarrow p \mathrm{ok}_t s$. ∎

Of immediate concern to a tester is what test method to use, or what test methods are better than others. An obvious and simple definition of what it means for M to be at least as good as N is that whenever N finds an error, so does M. More specifically, if a program tests incorrectly under N, it will also test incorrectly under M, with respect to the same specification. There is no particular reason why we should insist that M should use the same tests as N; we only need to know that *some* error is exposed by M whenever any error is exposed by N. Thus, within this notion of power of test methods lies the possibility that two methods perform entirely different tests and nevertheless are of equivalent power. This notion of power is formalized in the following definition.

Definition 4:

Given two test methods M and N, we say that M has *power greater than or equal to* N (denoted $M \geq N$) if $\forall p \forall s (p \mathrm{ok}_M s \Rightarrow p \mathrm{ok}_N s)$. ∎

While the simplicity of this definition is appealing, there are other possible definitions, most of which can be proved equivalent to this one. We will develop the formalism for one of these alternatives, one which will prove important at the end of this paper.

We begin by formalizing the idea of *reliability*. Intuitively, a test method M works reliably for p and s if we can rely on the results of the test that it suggests, $t = M(p, s)$. If p is correct with respect to s, then any test will say so, hence our only interest is in the results of tests when the program is incorrect. In this case, when p is incorrect with respect to s, the results of t must demonstrate this fact, otherwise M clearly cannot be relied upon for the particular program and specification in question. What we have said,

then, is that for M to be reliable for p and s, we must know that $\neg(p \text{ corr } s) \Rightarrow \neg(p \text{ ok}_M s)$, or in the contrapositive, $p \text{ ok}_M s \Rightarrow p \text{ corr } s$. Thus, we have

Definition 5:

A test method M is *reliable* for a set of programs P and a set of specifications S if we can infer the correctness of any program in P with respect to any specification in S by the results of the tests given by M. Formally we have a relation rel such that

$$p \text{ rel}_M s \Leftrightarrow (p \text{ ok}_M s \Rightarrow p \text{ corr } s).$$

Generalizing to sets of programs and sets of specifications,

$$P \text{ rel}_M s \Leftrightarrow \forall p(p \in P \Rightarrow p \text{ rel}_M s),$$

$$p \text{ rel}_M S \Leftrightarrow \forall s(s \in S \Rightarrow p \text{ rel}_M s),$$

and

$$P \text{ rel}_M S \Leftrightarrow \forall p \forall s(p \in P \wedge s \in S \Rightarrow p \text{ rel}_M s).$$

∎

The reliability relation is a characterization of a test method that is entirely independent of any algorithms embodied in the method, and of the actual tests selected by the method. Any two test methods that have exactly the same reliability relation are functionally identical. There is no reason to choose one over the other except, perhaps, for speed. For this reason, the relation seems to be an ideal basis for comparing the reliability or power of test methods. The most powerful test method imaginable would be one that performs reliably under all conditions, i.e., one that performs verification. Such a test method also has the largest imaginable reliably tested sets, P and S:

$$\mathsf{P} \text{ rel}_M \mathsf{S} \Leftrightarrow \text{ok}_M = \text{corr}.$$

This suggests that a measure of the power of a test method should be the size of the sets in its reliability relation. The following definition and theorm give us a characterization of this size.

Definition 6:

$$\text{ms}_M(P) = \bigcup_{S | P \text{rel}_M S} S$$

$$\text{mp}_M(S) = \bigcup_{P | P \text{rel}_M S} P$$

The function ms_M takes sets of programs to sets of specifications, denoting the *maximum reliably tested set* of specifications for M and P. mp_M is an analogous function from sets of specifications to sets of programs. ∎

Theorem I:

$$P \text{ rel}_M \text{ ms}_M(P)$$

and $\text{ms}_M(P)$ is the unique largest such set of specifications.

$$\text{mp}_M(S) \text{ rel}_M S$$

and $\text{mp}_M(S)$ is the unique largest such set of programs. ∎

What we have conjectured is that it is the size of the ms and mp sets that define the power of test methods. More specifically, we would like to say that M being more powerful than N means that $\text{ms}_M(P)$ is bigger than $\text{ms}_N(P)$ for all P, and the dual, that $\text{mp}_M(S)$ is bigger than $\text{mp}_N(S)$ for all S. This seems reasonable, because we are insisting that for M to be more powerful than N, M must be reliable for all the programs and specifications that N is, and more. It can be shown that power in this sense is well defined and that it is equivalent to the intuitive sense of power given in Definition 4.

Theorem II:

The following are equivalent:

$$\forall P(\text{ms}_M(P) \supseteq \text{ms}_N(P)),$$

$$\forall S(\text{mp}_M(S) \supseteq \text{mp}_N(S)),$$

and

$$M \geq N.$$

∎

An Application

We have seen that from the theoretical point of view, a test method is a function that depends on both programs and specifications. In order to be able to describe an algorithm for a test method in general requires that both programs and specifications be formal objects suitable for use as data in the computation that yields a test. Ever since the advent of compilers, programs have been suitable formal objects, but specifications are still rarely formal. For this reason, the most discussed and best understood test methods do not depend on specifications. Statement testing, mentioned above, falls in this category.

The restriction of test methods to depend on programs alone is a technical convenience only, so it comes as no surprise that the power of this class of test methods is also restricted. The usual argument for this is an informal one called the "missing path problem:" if the given program fails to perform an action required by the specification, then it may be that no amount of analysis of the program can produce test data that reveal the action's absence. Only by consulting the specification as well as the program can this be done. The loopholes in this reasoning are many, including the lack of definition of the term "action,"

how one is specified and how one can be missing. Nevertheless, the idea is clear and the missing path problem has driven all research to date on testing based on specifications.

One of the significant results of the mathematical framework introduced above is that the missing path problem can be formalized and broadly generalized.

A test method that depends only on programs is independent of specifications in the mathematical sense:

Definition 7:

A test method M is *specification independent* if $\forall p \forall s \forall r(M(p,s) = M(p,r))$. ∎

Specification independent test methods are sometimes called *structural* test methods, because they use the structure of the program to drive the generation of tests.

The fact that a test method is specification independent does not in and of itself limit the power of the method. In severly limited testing systems, even constant test methods may have the power of verification. If, however, we make a reasonable assumption about the generality of the underlying testing system, we find that specification independence does imply a limit on power. The necessary assumption is that for every program and test, we can always find a specification that will fool the test. In other words, we can always find a specification against which the program tests correctly, but with respect to which the program is incorrect. Among actual programs in any general purpose language, and among specifications robust enough to specify all programs, this is clearly true.

Theorem III:

For a specification independent test method M in a testing system that has the property that $\forall p \forall t \exists s(p \text{ ok}_t\ s \wedge \neg(p \text{ corr } s))$, we have $\text{mp}_M(S) = \phi$. ∎

This is not to say that specification independent testing is worthless. $\text{mp}_M(S)$ for some S smaller than S may well be nonempty.

The significance of this result is twofold. First, the limitation is expressed in the form of a restriction on the size of the reliably tested sets for the test method. Nowhere is there any appeal to program paths or specified actions. The limitation clearly applies in all programming environments, regardless of the nature or even the existance of paths in the programming language in use. Second, there is a duality in the framework between programs and specifications that makes it evident that an analogous theorem exists that limits the power of *program independent* or *functional* test methods. This frequently unrecognized fact suggests that testing practice should make use of tests generated *both* from programs and formal specifications.

Bibliography

[1] Geller, Matthew. "Test Data as an Aid in Proving Program Correctness." *Communications of the ACM*, 21 (May 1978).

[2] Goodenough, John B. and Gerhart, Susan L. "Toward a Theory of Test Data Selection." *IEEE Transactions on Software Engineering*, 1 (June 1975).

[3] Gourlay, John S. "A Mathematical Framework for the Investigation of Testing." *IEEE Transactions on Software Engineering*, 9 (November 1983). In press.

[4] Howden, William E. "Reliabililty of the Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering*, 2 (September 1976).