Empirical Evaluation of Test Coverage for Functional Programs

Yufeng Cheng*†, Meng Wang‡, Yingfei Xiong*†, Dan Hao*†, Lu Zhang*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
†Institute of Software, School of EECS, Peking University, China
‡School of Computing, University of Kent, U.K.
{chengyf, xiongyf, haodan, zhanglucs}@pku.edu.cn, M.W.Wang@kent.ac.uk

***Abstract*—The correlation between test coverage and test effectiveness is important to justify the use of coverage in practice. Existing results on imperative programs mostly show that test coverage predicates effectiveness. However, since functional programs are usually structurally different from imperative ones, it is unclear whether the same result may be derived and coverage can be used as a prediction of effectiveness on functional programs.**

In this paper we report the first empirical study on the correlation between test coverage and test effectiveness on functional programs. We consider four types of coverage: as input coverages, statement/branch coverage and expression coverage, and as oracle coverages, count of assertions and checked coverage. We also consider two types of effectiveness: raw effectiveness and normalized effectiveness. Our results are twofold. (1) In general the findings on imperative programs still hold on functional programs, warranting the use of coverage in practice. (2) On specific coverage criteria, the results may be unexpected or different from the imperative ones, calling for further studies on functional programs.

I. Introduction

Software developers need know how good the test suite on hand is in order to decide whether to stop testing because it is adequate, or continue with more. The quality of test suites is commonly defined by how well they are able to detect potential faults, known as *test effectiveness*, which can be measured in two ways: the basic *raw* effectiveness is the percentage of all defects that the test suite is able to detect, while *normalized* effectiveness [18] only concerns the defects that exist in the code that is actually executed by the test suite, resulting in a higher percentage.

Test effectiveness is hard to measure in practice and is commonly approximated by test coverage. Broadly speaking, test coverage can be divided into two categories: *input coverage* (concerning the percentage of code that is executed by certain test inputs) and *oracle coverage* (concerning the percentage of code whose effect on the output is examined by the test oracles). The former includes statement coverage, branch coverage, condition coverage etc. while the later includes checked/assertion coverage [28] and a simple count of the number of assertions [32]. Note that in this paper we consider only the traditional oracles on the output, but not the recently proposed inner oracles [31] on the internal state.

As an approximation, one natural question to ask is how closely test coverage is related to test effectiveness. There has been a lot of studies [6], [9], [10], [12], [13], [17], [18], [22], [29], [32] on imperative programs that mostly warrant the use of test coverage to predict test effectiveness. But giving the nature of test coverage (it targets source code instead of faults), the results are unavoidably sensitive to changes in program structure. It is known that even very simple program transformations, such as variable inlining, dramatically affect coverage measures [15], [26], and thus the effectiveness of test suites that achieve them.

The difference in code structure is far greater when comparing imperative programs to functional ones. For instance, a program written in the imperative style with iteration, state update and array is bound to be very different in structure from an equivalent version written in the functional style with recursion, immutable state and list. Then an immediate question is whether the empirical results based only on imperative programs remain applicable, which as far as we are aware, has not been answered. There is no study measuring test coverage against test effectiveness in a functional settings.

This lack of empirical evidence is certainly worrying, but was allowed to continue largely because traditionally functional languages, often developed for academically curiosity, did not cross path with mainstream software engineering. But the situation is changing rapidly. Companies including WhatsApp [1] and Facebook [25] now use functional languages in large scale. And ideas originated from functional programming such as lambda expressions, higher-order functions, list structure, generics, etc. have made their way into the design of many modern programming languages including Java, C++, C# and more. It is well conceivable that mainstream programmers may be using functional features in their code with or without realising. Therefore, it is more important than ever to reexamine the established software development practice

This work is supported by the National Basic Research Program of China under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61421091, 61432001, 61529201.

Yingfei Xiong is the corresponding author.

and see whether the same conclusions can be drawn for functional code.

In this paper we conduct the first empirical evaluation of test coverage for functional programs, and compare the results with those from imperative programs. One notable technical challenge is that, though there exist tools for measuring coverage for functional languages [3], [11], they do not use the standard criteria commonly found in mainstream software development, disabling cross comparison with imperative programs. To address this problem, we use a hybrid approach in our experiment. For standard statement/branch coverage, checked coverage, and the count of assertions, we map them to the functional setting closely following the principle of the design rational, rather than the letter of it, which is impossible due to the large differences between the language paradigms. On the other hand, we also consider expression coverage [11], a criterion that is specific to functional languages.

Based on the set of coverage criteria, we conduct two sets of experiments (concerning input and oracle coverages respectively) to measure the relation between test coverage and test effectiveness with Haskell [2] – the most popular modern functional language. We choose Haskell because it is known as a pure functional language, strictly free of imperative features such as side effects and assignment. Consequently, our result is "purely functional", allowing an interesting comparison with the imperative counterparts.

In summary, the contributions of the paper is listed as follows:

- New coverage criteria for functional programs, mimicking those of imperative programs.
- First experiment comparing input coverage with test effectiveness on functional programs.
- First experiment comparing oracle coverage with test effectiveness on functional programs.

II. How are functional programs different

Generally speaking, functional programming is a style of programming: the main program is a function that is defined in terms of other functions, and the primary method of computation is the application of functions to arguments. Unlike imperative programming, where computation is a sequence of transitions from states to states, functional programming has no implicit state and places its emphasis entirely on expressions (or terms). This style of programming is enabled and encouraged by a number of language features that were considered distinctively functional, but are increasing adopted by others [16].

A. Recursion and Pattern Matching

Functional programs are usually defined by recursion: functions call themselves in the definitions; and when the calls return, the results are used as components for constructing new return results. For instance, consider the insertion function of binary search trees. Figure 1 shows

```
function INSERT(k, T)
    root = T
    x = CREATE-LEAF(k)
    parent = NIL
    while (T != NIL) do
        parent = T
        if (k < KEY(T)) {
            T = LEFT(T)}
        else {
            T = RIGHT(T)}
    PARENT(x) = parent
    if (parent = NIL) {
        return x}
    else if (k < KEY(parent)) {
        LEFT(parent) = x}
    else {
        RIGHT(parent) = x}
    return root
```

Fig. 1. INSERT function in imperative style

```
insert k Empty = Node Empty k Empty
insert l (Node l x r)
    | k<x       = Node (insert k l) x r
    | k>=x      = Node l x (insert k r)
```

Fig. 2. INSERT function in functional style

an implementation in the imperative style, and Figure 2 is its functional equivalent in Haskell. In the Haskell code, the capitalised terms `Empty` and `Node` are *data constructors*, representing empty and non-empty trees respectively. When used on the left-hand side of a function definition, data constructors represent patterns that can be matched by input values; on the right-hand side, they are used in expressions to build values. Function `insert` has two clauses that deal with empty and non-empty trees respectively. When the tree is empty, a new node is created in its place; when the tree is non-empty, depending on the values of the inserted element and the current root (the boolean conditions `k<x` and `k>=x` that controls this are called *guards*), the element is recursively inserted either into the left or right subtree.

This programming style results in a very different (usually more succinct) code structure compared to the imperative style with iterative loops and explicit manipulation of pointers, which may require a different level of effort in achieving high code coverage. Indeed, to cover all the lines of this program, one test case alone is sufficient for the functional version, as long as the test traverses both left and right in the process of finding the inserting location, whereas we need at least three for the imperative version.

B. Higher-Order Functions

An idea of functional programming which turns out to be influential is higher-order functions—functions that take other functions as arguments and may produce functions as return results. Roughly speaking, a higher-order function captures a reusable recursion pattern over a data structure, and the functional arguments to it initialise the pattern into a concrete function.

For example, “fold” is a pattern that captures structurally inductive computation, where a function is apply to each structure layer and returns the accumulated result. Readers familiar with object orientation may think this as a variant of the VISITOR pattern. For example, we can define a function (`insertList`) that inserts a list of elements into a tree

```
insertList ks tree = foldr insert tree ks
```

which iterates through the key list and performs the insertions one by one, and returns the final tree with the inserted keys.

Traditionally considered a distinctive functional feature, higher-order functions are everywhere now. Google’s MapReduce is directly inspired by similar concepts in functional programming, and languages such as Java, C++ and C# jump one after another on the lambda-expressions bandwagon to streamline this kind of higher-order programming.

In the context of testing, the use of higher-order functions further distant functional code from its imperative counterpart. Due to the reuse, functions like `insertList` have extremely short definitions, which puts the effectiveness of statement coverage in doubts, as it becomes trivial to cover the program, including all the library functions it calls!

C. Immutable Data and Lazy Evaluation

Another important characteristic of functional programming that has impact on testing is the absence of destructive updates, known as *referential transparency* or *purity*. In the functional code in Figure 2, the symbol ‘=’ means true equality: `insert k Empty` is equal to `Node Empty k Empty` in any execution context, and can be used interchangeably. This is very different from the case in the imperative sense, where ‘=’ refers to a destructive update, assigning a new value to the left-hand-side variable. Consequently, there is no concept of system state in pure functional programming; a function’s behaviour is completely determined by its definition and the arguments passed to it.

In testing, this means there is less issue with observability, as the returned value is the only effect of execution. The case of oracle coverage becomes interesting when we consider Haskell’s call-by-need evaluation strategy (also known as *laziness*), which delays the evaluation of an expression until its value is needed. For example, if we only need the head of a list, then the expressions that computes the rest of the list structure will not be evaluated. With this feature, we can straightforwardly create program slices that are evaluated by selectively ‘using’ parts of the returned value to simulate checked coverage.

III. Coverage Criteria for Functional Programs

With the very different language constructs, functional languages do not naturally share the same kind of coverage criteria developed in the imperative world. Typically, functional programs are based on expressions instead of statements, and pattern matching (together with boolean guards) are more common than `if`-branching. As a result, one shall either try to map traditional criteria to functional constructs (which may or may not be possible), or devise new ones for the functional setting. For the study in this paper, we consider both cases. Specifically, we map the usual statement/branch coverage and checked coverage to Haskell, and use a notion of expression coverage that is native to Haskell.

Statement/Branch Coverage. Since there is no construct in Haskell that is equivalent to imperative statement, we take an approximation to consider units of computation that typically occupies a line of code. This means each clause of a function defined with pattern matching, which is also known as an *alternative*. For example, in Figure 2 there are three “statements” in the definition of `insert` concerning the empty-tree cases and the two alternatives of the non-empty-tree case. If there are local definitions (i.e., functions defined in `let` expression or in `where` expression), we count their clauses as separate statements too. A case that is difference from the imperative setting is `if`-conditionals: in Haskell both branches have to be non-empty (to explicitly pass on control) and branch coverage (each branch condition must have been true at least once and false at least once) now coincide with statement coverage. A very similar argument applies to guards: a clause’s pattern must fail to match for the next clause to be tried (just like nested `if`s), and if all clauses fail to match the program simply crashes. For this reason, in our experiment we consider statement and branch coverage as one, and write statement/branch coverage.

Expression Coverage. Expression coverage [11] is a coverage criteria used in testing functional programs. A Haskell program can be viewed as a set of expressions, where each expression is composed from smaller expressions, i.e., an abstract syntax tree of expressions. We use N to denote the total number of expressions, including both the primitive expressions and expressions composed from other expressions, and use E to denote the total number of expressions where at least one of its sub expression (including itself) is evaluated. Then expression coverage is defined as E/N. Expression coverage is more refined than statement coverage because of lazy evaluation: a statement may be evaluated but not necessarily all of its expression components.

Count of Assertions. Count of assertions is a simple measurement of oracle coverage used in existing empirical studies [32]. Assuming that more assertions indicates better oracles, we can use assertion counts to predicate effectiveness. However, functional programs do not tend include a variety of assertions; instead oracles are typically expressed as equations involving the final returned value of a function—this is adequate as the returned value is the only effect of execution. In our experiment, we divide such all-inclusive equations into ones concerning individual components of the returned value, to populate the number of (virtual) assertions and therefore enable counting. For example, data constructed by constructor `X a b` has two components, corresponding to `a` and `b`, respectively. Similarly a list of size n contains $n + 1$ components. Each of the first n component corresponds to an item in the list, and the last component is the size of the list.

Checked/Assertion Coverage. Checked coverage [28], sometimes referred as assertion coverage [32], is a way to measure how much the test oracles cover the original program. Checked coverage is measured by performing a backward slicing on the programs, and measure how many statements are included in the backward slice. To map checked coverage to functional program, we use the same definition of "statement" as in statement/branch coverage to count the number of statements in the backward slices.

IV. Methodology

We will answer three research questions in this paper:

RQ1. Is input coverage correlated with effectiveness in functional programs?

RQ2. Is oracle coverage correlated with effectiveness in functional programs?

RQ3. How are the correlations compared with those obtained from imperative programs?

Since many publicly-available Haskell programs do not come with test cases, we resolve to automatic generation of test inputs and assertions for the selected programs. We also mutate the programs to get variants with injected faults. Finally, we execute the programs (and variants) to measure coverages and effectiveness. In the rest of the section we explain the details of this process.

Our experiments were carried on a Linux virtual machine with 8 GiB memory. The host machine has an eight-core Xeon E5-1410 CPU. Our experimental programs and data are available online at https://github.com/onetwogoo/coverfun.

A. Subject Programs

We selected the standard NoFib [24] benchmark suite as our subject programs. NoFib is a widely-used benchmark suite in evaluating Haskell compilers [19]. The programs in NoFib are divided into three subsets, Real, Imaginary, and Spectral. The Real subset consists of real-world programs written in Haskell. Typical programs in the Real subset include a Prolog interpreter, a theorem-prover, an arbitrary-precision calculator, a Fluid-dynamics program, etc.

We rule out programs that use non-standard Haskell features (such as the C preprocessor) and select 24 programs from Real, totalling 25k lines of code (excluding comments and empty lines). The names of the programs and the lines of code are shown in the first two columns in Table I. Note that, as shown by a study [23] Haskell programs usually have much fewer lines of code than their counterparts in C or Java.

B. Generating Test Input

Since the NoFib programs come without test cases, we use Irulan [4] as a tool to generate test inputs. Compared to other generation methods [7], [27], Irulan can automatically generate test inputs without requiring generator functions for user-defined datatypes. In addition, Irulan has a proven ability of generating tests achieving high coverage, a feature that is very useful in our study.

For each program in our experiment, we run Irulan to generate test inputs for it. To control the running time of our experiment, we randomly select for each program 1000 test inputs. We omit functions that produce as return values functions or structures containing functions, because functions by nature cannot be compared for equality (a feature needed to construct oracles). Nevertheless, this does not exclude us from testing higher-order programming since such high-order functions are used in other functions in final executions and are tested indirectly. Similarly, we do not generate function values as test inputs either, and rely on indirect testing for higher-order functions that require such arguments.

The numbers of test inputs are shown in "#Cases" column Table I. In a few exceptional cases, when the program is dominated by higher-order functions that return function values, we may end up with less than 1000 cases. Column "#Tested Func." shows the number of functions we tested and column "#Func." shows the total number of functions. Finally, "Stmt/Brch. Cov." column and "Expr. Cov." column show the statement/branch coverage and expression coverage of the test input on the programs, respectively.

C. Generating Test Oracles

Since we use the original program as a correct version and inject faults into the program, we can directly use the output of the original program as test oracle. To effectively measure output coverage, we additionally divide the output into components and generate assertions for each as described in Section III.

D. Generating Faults

To measure the effectiveness of a test suite, the usual way is to use a set of faulty programs and check how many faults can be revealed by the test suites. Since we do not have a set of faulty versions of NoFib programs, we inject faults by mutating the programs. An existing study [20] showed that faults injected by mutation operators can well simulate real faults.

TABLE I
Basic Statistics on Subject Programs

Program	#Lines	#Cases	#Mutants	#Faults	#Func.	#Tested Func.	Stmt/Brch. Cov.	Expr. Cov.	#Assertions
anna	5289	1000	948	296	534	379	69%	43%	2702
bspt	1284	1000	893	400	224	136	48%	59%	4334
cacheprof	1409	461	383	194	141	23	4%	10%	1343
fem	754	1000	942	116	114	19	19%	14%	3578
fluid	1470	1000	947	184	135	31	33%	23%	2933
fulsom	907	1000	928	470	267	146	64%	71%	3708
gamteb	468	1000	985	782	60	28	84%	82%	2962
gg	824	1000	845	292	140	100	46%	46%	2226
grep	263	1000	240	27	46	21	79%	67%	2241
hidden	442	1000	515	188	48	25	48%	45%	2632
hpg	961	1000	903	95	157	66	31%	14%	2056
infer	746	1000	298	130	157	53	66%	55%	2105
lift	495	1000	584	310	74	47	75%	69%	2952
linear	1023	1000	757	245	257	51	33%	33%	2387
maillist	81	649	125	15	10	4	26%	11%	1557
mkhprog	198	1000	192	42	31	11	23%	25%	2032
parser	751	432	158	24	81	53	31%	27%	1950
pic	305	1000	604	485	26	7	87%	86%	3982
prolog	427	1000	546	61	63	19	28%	26%	1993
reptile	1041	1000	880	453	242	152	57%	54%	2393
rsa	64	1000	152	113	13	3	84%	82%	2018
scs	613	1000	789	41	65	15	16%	12%	1830
symalg	863	1000	928	303	73	33	48%	47%	2276
veritas	4193	1000	921	228	673	198	42%	34%	2172
Total	24871	22542	15463	5494	3631	1620	47%	43%	60542

We use the open source tool MuCheck [21] to generate mutants. MuCheck is similar to many mutation testing tools that apply a set of predefined mutation operators to the source code. We configure MuCheck parameters to generate mutants in an efficient and fair manner: (1) We assign lower weights to operators that mainly generate equivalent mutants in the functional setting. Specifically, we set `doMutatePatternMatches` and `doMutateValues` parameters in MuCheck to a low value (0.5), after observing large amount of equivalent mutants generated by them. (2) We control the order of application so that the results are not dominated by the a few operators that tends to generate a lot of mutants. More specially, we first apply the operators that tend to generate fewer mutants and then apply the operators that tends to generate more mutants.

We generated 200 mutants for each module. However, since each program consists of several modules, the number of mutants may be too large in some cases; and thus we further control the number by randomly selecting 1000. We also remove mutants that change the interface of the program and fail to compile. The final number of mutants is shown in the "#Mutants" column in Table I.

We consider a mutant equivalent (not a fault) if it is not killed by any of our generated tests. The number of faulty versions obtained through the above process is shown in the "#Faults" column in Table I.

E. Executing Tests

Haskell is evaluated lazily. So no execution happens unless the return value of a function is demanded. To create this demand, we print the returned values and force the evaluation. An issue with this approach is infinite data structure: it is idiomatic Haskell to write data producing functions that potentially go on forever and generate infinite steams of data, with the expectation that consumption of the data will be finite and thus the whole program terminates. Since there is no way of telling an infinite structure from finite ones (the equivalent of the Halting problem), we take an programatic approach by truncating all outputs to 1024 characters. The assertions we generate are also based on the truncated outputs.

We consider a mutant killed by a test case if it crashes or times out during execution, or it fails the test oracle.

F. Measuring Input Coverage

As mentioned before, we consider two types of coverage, namely statement/branch coverage and expression coverage. To measure coverage, we use the Haskell coverage tool HPC [11], which instruments Haskell programs, runs the instrumented programs, and then records coverage measures. HPC supports three coverage criteria specifically designed for Haskell, which are (1) clause coverage: the coverage of clauses in functions defined with pattern matching, (2) boolean coverage: branch coverage involving boolean expressions used in guards and `if` conditions, and (3) expression coverage: the same expression coverage that we use in our experiments. As a result, we can compute statement/branch coverage by combining the clause and boolean coverages, and directly obtain the result of expression coverage.

G. Measuring Oracle Coverage

Following from Section III, we approximate the count of assertions by the number of output components that are checked. Since each assertion we generate checks exactly one component, we can simply count the number of output components that are involved in an oracle. The count of assertions of each program is shown in "#Cases" column of Table I.

Checked coverage is normally measured through backwards slicing, working out the percentage of the program that is involved in producing the output component. As discussed in Section II-C, Haskell's lazy evaluation comes very handy here. We do not need a separate slicing tool other than HPC because by fixing the output component demanded by an assertion, only relevant expressions will be evaluated and recorded as standard expression coverage. We do not include the maximum checked coverage (achieved when full oracles are used) in Table I because it is the same as the maximum statement/branch coverage already shown.

H. Measuring Effectiveness

We measure two types of test suite effectiveness. Raw effectiveness is the percentage of faults that a test suite finds. Normailized effectiveness is the percentage of *covered* faults that a test suite finds. Here we say a fault is covered by a test suite if that test suite executes the function where the fault lies. Normalized effectiveness is proposed by Inozemtseva and Holmes [18] to measure the ability of test suites to trigger and detect a fault, rather than to cover the faulty statements. Their study [18] has found out that input coverage criteria do not have a strong correlation with normalized effectiveness.

I. Data Analysis

To conduct the experimental efficiently, we first run all test inputs on all programs and examine all test oracles. In this process, we record the coverage information of each test input and each generated assertion, and whether each pair of test input and assertion captures a fault. Then based on the data, we simulate the results of running different tests with different levels of coverage. Our simulation aims to repeat the experimental procedures in existing empirical studies on imperative programs [18], [32], so that the result can be compared with the existing results.

To answer the first research question, we randomly group individual test cases into suites of fixed sized of 3, 10, 30, 100, and 300. For each size, 100 suites are generated and their coverages and the two effectiveness values are computed.

To answer the second research question, we randomly select assertions for each test input to form test oracles of different strength. For the counts of assertions, we randomly select $n\%$ assertions from the original set for each test. For checked coverage, we randomly select assertions until $n\%$ of the maximum checked coverage for the test input is reached. We use five different n values: 20, 40, 60, 80, and 100. In this research question we only consider raw effectiveness as only raw effectiveness is considered in existing work on imperative programs.

To answer the third research question, we compare our result with the results from two most recent empirical studies [18], [32] on imperative programs.

V. Results and Analysis

We say a correlation to be *strong* if the correlation coefficient is more than 0.8, *moderate* if the coefficient lies between 0.5 and 0.8, *weak* if the coefficient lies between 0.3 and 0.5, or *very weak* otherwise.

A. Is input coverage correlated with effectiveness in functional programs?

The result of the analysis is shown in Figure 3 and 4. We present the graph in the same visual form as the study by Inozemtseva and Holmes [18] to enable cross comparison. Each row shows the results of a program and each column shows the results of a test suite size, while the last column is for all sizes. Each small graph has coverage as its x axis and effectiveness as its y axis. Different coverage oracles are represented as different colors, where red represents statement/branch coverage and blue represents expression coverage.

To quantitatively understand the correlation, we also compute the Kendall τ correlation coefficients between two types of coverage and two types of effectiveness, as shown in Table II and III. Due to space limit, we include only the coefficients for the combined results, and those for each fixed test suite size are available online.

First let us consider normalized effectiveness. As we can see from Figure 3, when the coverage increases, no consistent change of effectiveness can be observed. As a matter of fact, one coverage value usually corresponds to a wide range of effectiveness value. Also, the coefficients in Table II are generally smaller the 0.8. The results from both statement/branch coverage and expression coverage are similar. These observations indicate the following finding.

Finding 1: Input coverage has no strong correlation with normalized test effectiveness.

This result is expected: normalized effectiveness concerns the ability to trigger and capture a fault, and these abilities are largely decided by test oracles but not test input.

Second let us consider raw effectiveness. As we can see from Figure 4, the dots mostly form a tilted line pointing from the lower left corner to the upper right corner. Also, the coefficients in Table III are generally larger than 0.8, with an average value 0.87 and a minimum 0.74. From both the figure and the table, there is no significant difference between statement/branch coverage and the expression coverage. These observations indicate the following finding.

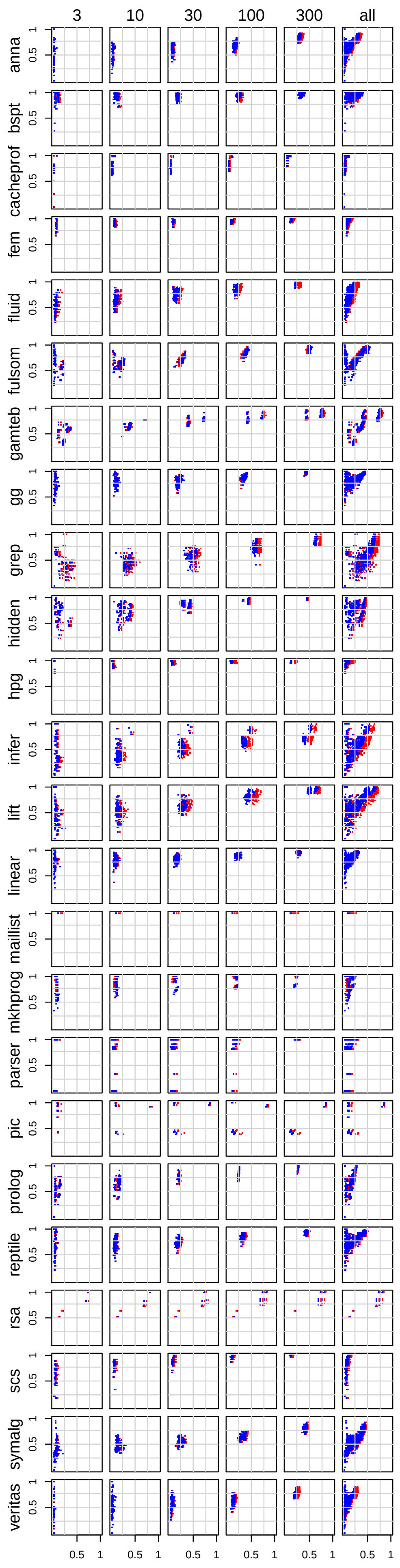

Fig. 3. Coverage and Normalized Effectiveness

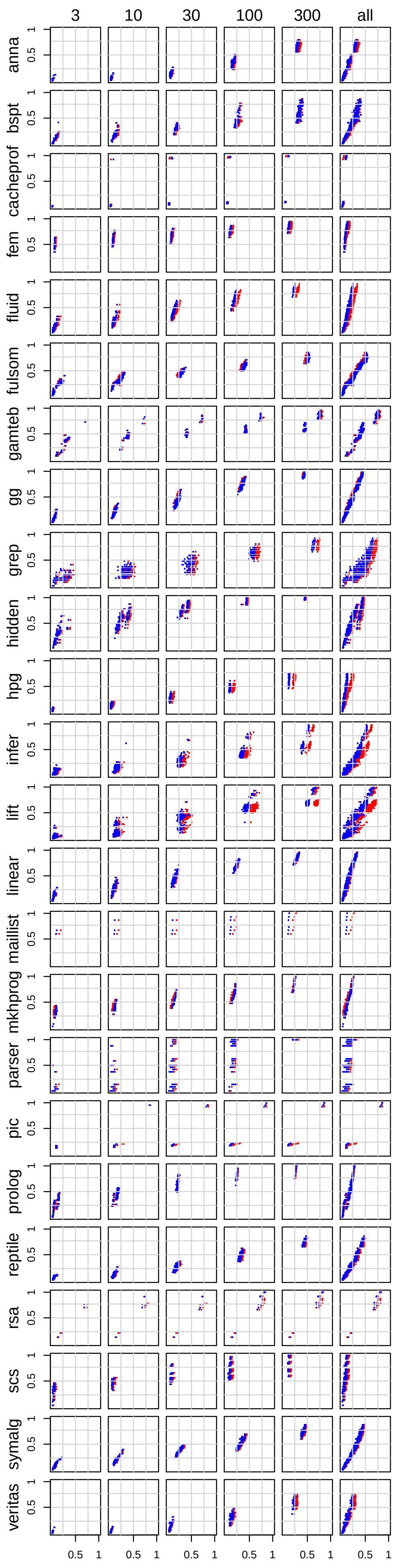

Fig. 4. Coverage and Raw Effectiveness

TABLE II
The Kendall τ coefficients between different types of coverage and normalized effectiveness

Program	Statement/Branch	Expression
anna	0.51	0.52
bspt	0.08	0.10
cacheprof	0.54	0.59
fem	0.26	0.28
fluid	0.68	0.67
fulsom	0.57	0.57
gamteb	0.77	0.79
gg	0.42	0.42
grep	0.60	0.61
hidden	0.59	0.60
hpg	-0.26	-0.27
infer	0.48	0.51
lift	0.65	0.63
linear	0.46	0.46
maillist	0.00	0.00
mkhprog	0.14	0.15
parser	0.66	0.67
pic	-0.29	-0.28
prolog	0.78	0.77
reptile	0.47	0.48
rsa	0.90	0.91
scs	0.79	0.76
symalg	0.63	0.63
veritas	0.41	0.41

TABLE III
The Kendall τ coefficients between different types of coverage and raw effectiveness

Program	Statement/Branch	Expression
anna	0.87	0.88
bspt	0.86	0.88
cacheprof	0.89	0.92
fem	0.88	0.89
fluid	0.88	0.89
fulsom	0.89	0.89
gamteb	0.85	0.88
gg	0.91	0.92
grep	0.81	0.80
hidden	0.84	0.84
hpg	0.88	0.85
infer	0.86	0.86
lift	0.78	0.80
linear	0.89	0.90
maillist	0.97	0.95
mkhprog	0.88	0.91
parser	0.73	0.74
pic	0.89	0.89
prolog	0.92	0.90
reptile	0.90	0.91
rsa	0.93	0.94
scs	0.75	0.78
symalg	0.92	0.92
veritas	0.87	0.87

Finding 2: Input coverage has a strong correlation with raw test effectiveness.

Third, by comparing statement/branch coverages and expression coverage in Table I, Table II, Table III, we have the following finding. This finding shows that though expression coverage is more fine-grained, it does not provide significant boost over statement/branch coverage.

Finding 3: Statement/branch coverage and expression coverage have no significant difference in their correlation with test effectiveness.

B. Is oracle coverage correlated with effectiveness in functional programs?

The results are shown in Figure 5 and Figure 6. Again, we present the result in the same way as existing work [32] to enable cross comparison. The x axis is the count of assertions or checked coverage and the y axis is raw effectiveness. The raw effectiveness is plotted as an error bar for each coverage level.

We make the following observations from the figures. First, as we can see from the figures, both count of assertions and checked coverage is correlated to test effectiveness. When the count of assertions or the checked coverage increases, the test effectiveness increases. However, the correlations between the two coverage criteria are different. Checked coverage usually has large Kendall coefficients, with most larger than 0.8, while counts of assertions have only small Kendall coefficients, between 0.21 and 0.33. As a result, we have the following findings.

Finding 4: Count of assertions has very weak correlation with test effectiveness.

Finding 5: Checked coverage has strong correlation with test effectiveness.

The findings indicate that for functional programs, we cannot just check the coverage of oracles on the output, but have to understand more deeply on what part of the program produces the output.

C. How are the correlations compared with those obtained from imperative programs?

Among our five findings, Findings 1, 2, 4, and 5 can be compared with existing studies [18], [32] on imperative programs. Among them, Findings 1, 2, and 5 are consistent with those on imperative programs, where those studies find or deny a strong correlation with similar Kendall coefficient values. The only exception is Finding 4, where Zhang and Mesbah found that the count of assertions is strongly correlated with test effectiveness on imperative programs. We suspect the reason is the difference between imperative programs and functional programs. Imperative programs usually have a large final state to assert, where a new assertion usually checks a new distinct part of the state. On the other hand, functional programs usually have smaller output. Though the output may contain data structure such as lists or trees, different parts of the returned data are likely to be produced by the same part of the program, and thus checked coverage plays a more important role in functional programs because it deeply investigate the coverage on the program but not only the output.

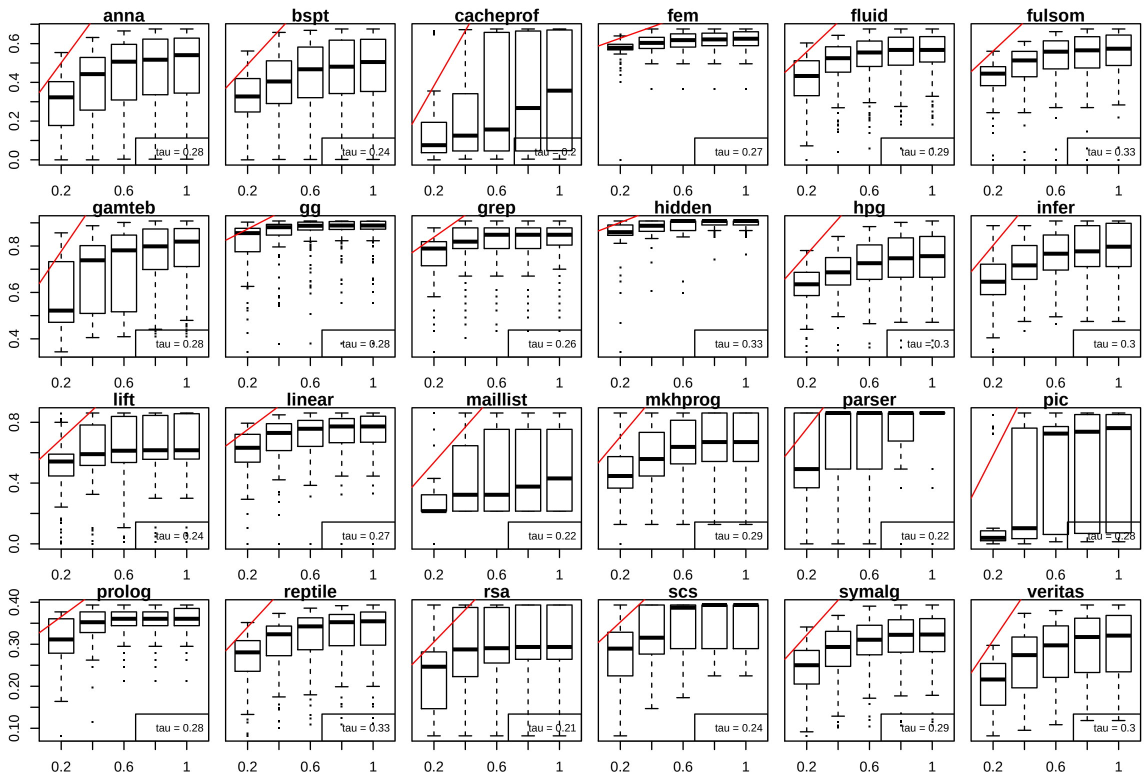

Fig. 5. Count of Assertions and Raw Effectiveness

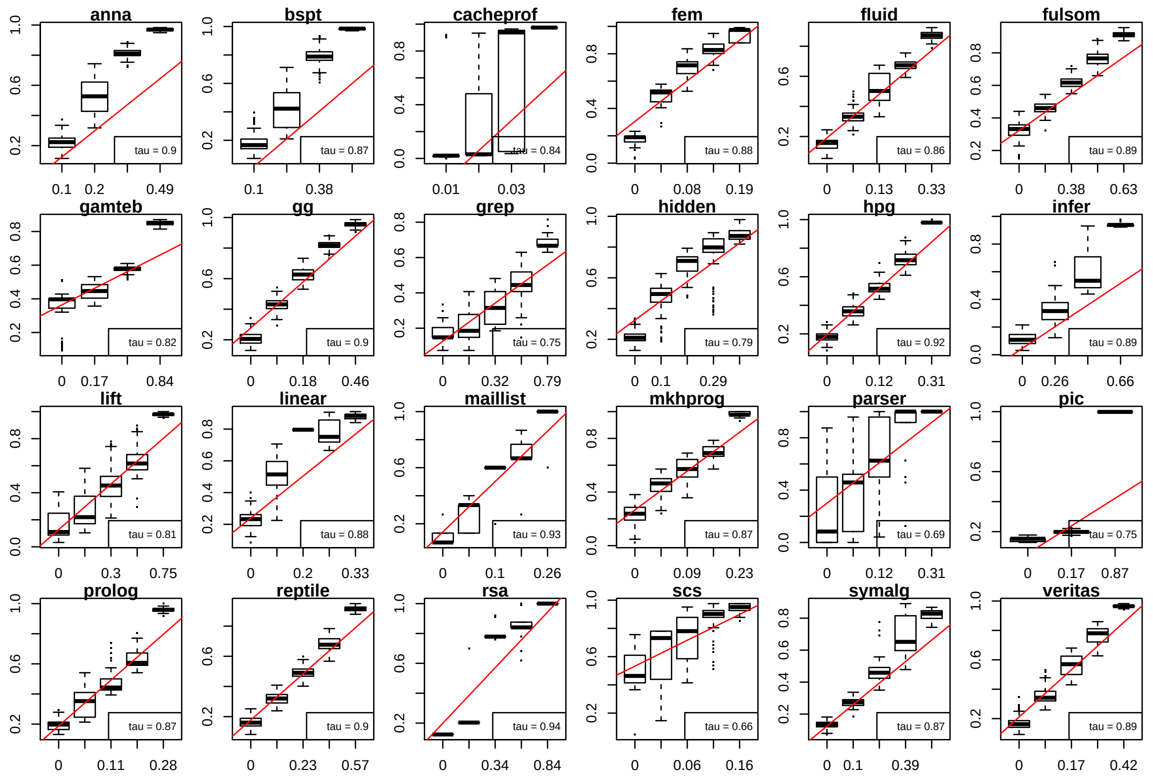

Fig. 6. Checked Coverage and Raw Effectiveness

VI. Threats to Validity

The automatically generated test inputs achieve low coverage in some subject programs. The correlation may be different from the correlation of high coverage test inputs for those programs.

We use automatically generated test inputs and assertions. According to existing research [13], [32], the correlation between generated inputs and effectiveness tends to be weaker than that of manually written tests. Therefore, the correlation may be stronger when we consider manual test input and oracles.

We use mutants as a simulation of real faults. Although there are many studies supporting it in imperative languages [5], [6], [8], [20], whether or not it is a good simulation leaves uncertain in functional languages. More research is needed to clarify this issue.

VII. Related Work

A lot of existing studies have contributed to the correlation between effectiveness, test suite size, input coverage, and oracle coverage.

Frankl et al. [10] compared the effectiveness of the all-uses and all-edges test data adequacy criteria. Hutchins et al. [17] investigated the effectiveness of all-edges and all-DUs coverage adequacy criteria for fault detection. Frankl [9] did an empirical evaluation of the fault-detecting ability of decision coverage and the all-uses data flow testing criterion. Andrews et al. [6] compared block, decision, c-use and p-use coverage criteria with mutation analysis. Namin et al. [22] studied the relation between test suite size, structural coverage, and fault-finding effectiveness. Shin et al. [29] evaluated the fault detection effectiveness of Function Block Diagram (FBD) model-based test coverage criteria. Gligoric et al. [12] presented an extensive study that evaluates coverage criteria over non-adequate suites. Gopinath et al. [13] used suites from a large set of real-world open-source projects to determine which coverage criteria best predict mutation kills.

Inozemtseva et al. [18] studied the correlation between coverage and effectiveness. First, they picked five large Java programs from application domains as their subjects. Second, they used PIT to generate faulty versions of the program. Then, they chose 7 test suite sizes with each size containing 1000 randomly selected test suites. They measured statement, decision and modified condition coverage of all test suites. To measure the effectiveness of these test suites, they computed their normalized mutation score which means the number of mutants a test suite kills divided by the number of mutants it covered. Finally, they analyzed their data and found there is a moderate to high correlation between coverage and effectiveness but only a low to moderate correlation when the number of test cases in the suite is controlled for.

Zhang et al. [32] investigated the correlation between assertions and test suite effectiveness with five Java programs of different sizes. First, they studied the influence of the number of assertions by generating test suites in three ways: randomly, controlling test suite sizes, and controlling number of assertions. In all three cases, they found there is a strong correlation between the effectiveness of test suite and the number of assertions it contains. Second, they defined assertion coverage as the fraction of statements in the source code executed via the backward slice of the assertion statements in a test suite. They compared the correlation between assertion coverage, statement coverage and effectiveness of randomly generated test suites and of test suites controlling assertion coverage. They found effectiveness is strongly correlated to assertion coverage and is more sensitive to assertion coverage than statement coverage. Finally, they analyzed the influence of the type of assertions by comparing test suites of different assertion categories. They found different types of assertions can influence the effectiveness of their containing test suites.

To the best of our knowledge, there is no research at the time of writing about the correlation between coverage and effectiveness in functional programs.

Besides the input coverage and oracles coverage criteria considered in this paper, researchers have also proposed other types of coverages and oracles. For example, Dan et al. [14] proposed to generate tests to cover "suspicious" statements. Xiong et al. [31] proposed to use inner oracles that assert the internal states of programs rather than just the output. Tao et al. [30] proposed a coverage criteria that assigns lower weights to loop guards than to other branches. It remains as future work to study the relation between these criteria and test effectiveness.

VIII. Conclusion

In this paper, we studied the correlation between test coverage and test effectiveness in functional programs. Our results indicate that most findings on imperative programs are still valid on functional programs, asserting the continuous use of coverage in practice. Our results also show that count of assertions, which is shown to be a good indicator of effectiveness on imperative programs, has very weak correlation to effectiveness on functional programs. Also, statement/branch coverage and expression coverage do not have significant difference on functional programs. These results indicate that though the use of coverage on functional program is promising, we still need more fine-grained study to understand how different coverage criteria work on functional programs.

References

[1] https://www.erlang-solutions.com/about/news/erlang-powered-whatsapp-exceeds-200-million-monthly-users.

[2] https://www.haskell.org/.

[3] E. AB. Cover: A coverage analysis tool for Erlang. http://www.erlang.org/doc/man/cover.html.

[4] T. Allwood, C. Cadar, and S. Eisenbach. High coverage testing of Haskell programs. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 375–385. ACM, 2011.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411. ACM, 2005.

[6] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[8] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 158–171. ACM, 1996.

[9] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. *ACM SIGSOFT Software Engineering Notes*, 23(6):153–162, 1998.

[10] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[11] A. Gill and C. Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–12. ACM, 2007.

[12] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[13] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82. ACM, 2014.

[14] D. Hao, L. Zhang, M.-H. Liu, H. Li, and J.-S. Sun. Test-data generation guided by static defect detection. *Journal of Computer Science and Technology*, 24(2):284–293, 2009.

[15] M. Heimdahl, M. Whalen, A. Rajan, and M. Staats. On MC/DC and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 5.B.3–1–5.B.3–13, Oct 2008.

[16] Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[18] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[19] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[21] D. Le, M. A. Alipour, R. Gopinath, and A. Groce. Mucheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.

[22] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth International Symposium on Software Testing and Analysis*, pages 57–68. ACM, 2009.

[23] S. Nanz and C. A. Furia. A comparative study of programming languages in Rosetta code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 778–788, Piscataway, NJ, USA, 2015. IEEE Press.

[24] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

[25] C. Piro and E. Letuchy. Functional programming at Facebook. In *Commercial Users of Functional Programming Conference*, 2009.

[26] A. Rajan, M. W. Whalen, and M. P. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 161–170, New York, NY, USA, 2008. ACM.

[27] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM.

[28] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[29] D. Shin, E. Jee, and D.-H. Bae. Empirical evaluation on FBD model-based test coverage criteria using mutation analysis. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 465–479. Springer Berlin Heidelberg, 2012.

[30] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao. Cooperative software testing and analysis: Advances and challenges. *Journal of Computer Science and Technology*, 29(4):713–723, 2014.

[31] Y. Xiong, D. Hao, L. Zhang, T. Zhu, M. Zhu, and T. Lan. Inner oracles: Input-specific assertions on internal states. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 902–905. ACM, 2015.

[32] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 618–624. ACM, 2015.