Extending Coverage Criteria by Evaluating Their Robustness to Code Structure Changes

Angelo Gargantini, Marco Guarnieri, Eros Magri

Extending Coverage Criteria by Evaluating their Robustness to Code Structure Changes

Angelo Gargantini, Marco Guarnieri, and Eros Magri

Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{angelo.gargantini,marco.guarnieri,eros.magri}@unibg.it

Abstract. Code coverage is usually used as a measurement of testing quality and as adequacy criterion. Unfortunately, code coverage is very sensitive to modifications of the code structure, and, therefore, the same test suite can achieve different degrees of coverage on the same program written in two syntactically different ways. For this reason, code coverage can provide the tester with misleading information.
In order to understand how a testing criterion is affected by code structure modifications, we introduce a way to measure the sensitivity of coverage to code changes. We formalize the modifications of the code structure using semantic preserving code-to-code transformations and we propose a framework to evaluate coverage robustness to these transformations, extending actual existing coverage criteria.
This allows us to define which programs and which test suites can be considered robust with respect to a certain set of transformations. We can identify when the obtained coverage is fragile and we extend the concept of coverage criterion by introducing an index that measures the fragility of the coverage of a given test suite. We show how to compute the fragility index and we evidence that also well-written industrial code and realistic test suites can be fragile. Moreover, we suggest how to deal with this kind of testing fragility.

1 Introduction

The notion of code coverage and testing criteria dates back to the early sixties [14,19]. Although, as Dijkstra claimed in 1972 [6], "program testing can be used to show the presence of bugs, but never their absence", the initial research focused on finding *ideal* testing criteria, i.e. those capable, under certain assumptions, of demonstrating the absence of errors in programs by testing them [10]. Researchers soon realized that finding such ideal testing criteria was impractical if not impossible [23] and the community started to introduce, compare, and study testing criteria, which are not ideal but they have proved to be useful to measure the quality of testing and to find faults in programs. However, there still exists some skepticism around the actual significance of coverage criteria. It is well known that some faults may be completely missed by test suites adequate to some coverage criteria (for instance statement coverage cannot discover omission faults) and that testing criteria are very sensitive to the structure and to the syntax of the code, regardless its actual behavior. Rajan et al. [17] show that

the MCDC, required by FAA for software on commercial airplanes, and often considered a very tough criterion to achieve, can be easily cheated by simple folding/unfolding of conditions inside guards.

Despite their weakness, coverage criteria give an immediate and easy to compute (at least for simple criteria) feedback about the quality of the testing activity. Once a test suite has been developed or built, one wants to know which parts of the code are exercised and which are not, and this information can be simply obtained by running the code with the tests. Coverage is often used as acceptance threshold: if a test suite achieves a given coverage, it is considered adequate, and the tested software accepted as *good*. For this reason, reaching a given level of coverage becomes a critical factor during the testing activity.

We can state that the coverage data are easily obtained and are widely used as acceptance measure, but have a questionable significance. In this paper we try to augment the information one can retrieve from the coverage data by considering also the structure of the code and its possible transformations.

There are two main scenarios in which it is important knowing how the coverage offered by a test suite behaves with respect to the changes in the code structure.

A. The code has been transformed *in the past* before being tested and the coverage may depend on the transformations applied. In this way, testing criteria can easily be cheated, and hence an additional measure of the coverage fragility helps in identifying well-tested classes from poorly tested ones.
B. The code structure will change *in the future* without changing the semantics of the program by applying some refactoring rules, by some automatic transformations, or by introducing particular patterns. This may influence the coverage after the application of these transformations. In this context, the tester would like to know if the level of coverage provided by the test suite will be preserved, i.e., if the coverage is robust w.r.t. future changes.

In this paper we make use of code transformations that preserve code functional behavior to model code changes. We formally define when a test suite achieves a *fragile* vs *robust* coverage. Roughly speaking, a fragile coverage depends on the structure and syntax of the code, and the test suite would not achieve the same level of coverage on the transformed code. A robust coverage and a robust test suite do not suffer from modifications of the code structure. We introduce a measure of fragility by extending the usual coverage criteria.

In Section 2 we present some related work on testing, coverage, and code transformations. Section 3 presents the theoretical framework in which our work will be integrated and contains some examples of useful code transformations. In Section 4 we show the limitations of actual coverage criteria in terms of their fragility with respect to code changes by giving several examples in which the obtained coverage is fragile. We introduce and formally define the concept of coverage robustness and several measure of coverage fragility. Section 5 reports some experiments. In Section 7 we present our conclusions.

2 Related Work

The concept of code coverage was first clearly introduced by Miller and Maloney from the US Chemical Corps in 1963 [14], although some similar concepts were already introduced by Senko from IBM [19]. Miller and Maloney observed that it is not sufficient to know that a program passes all the tests, since each test case checks a portion of the program and some portions may be not tested at all. They developed a model of programs based on flow charts and logical trees and required that every case is tested at least once. Following their approach, various notions of code coverage have been proposed as a measure for test suite quality, including statement coverage, branch coverage, method coverage, MCDC, and others [15]. These criteria consider the code structure, give a measure of the adequacy of the testing activity, and they can be used to drive the testing activity itself, for instance, requiring that certain tests must be generated. These criteria do not guarantee that the program is correct if it passes adequate testing, i.e. they are not *ideal* [10]. However, they have practical utility and they are generally required for commercial software. For instance, the Modified Condition Decision Coverage (MCDC) [5], is required for safety critical aviation software by the RCTA/DO-178B standard. The basic assumption is that a test suite is likely to be effective at revealing faults if it exercises the code where the fault is located. Therefore, increased code coverage is expected to correlate with more revealed faults, although other factors may influence the actual outcome [16]. Staats et al. [20] show that test suites generated specifically to satisfy coverage criteria achieve poor results in terms of effectiveness, whereas the use of coverage criteria as a supplement to random testing provides an improvement in the effectiveness of the generated test suites.

It is well known that coverage criteria can be very sensitive to code structure both if they are used for measuring test adequacy and if they are used for test generation. Regarding the adequacy, there are several works arguing that code coverage is not robust to code structure transformations. In [13], the authors show how very simple transformations (like adding a new empty line) can confuse code coverage tools. More severe issues are presented in [17]. In that paper, the authors prove that the MCDC metrics is highly sensitive to the structure of the implementation and can therefore be misleading as a test adequacy criterion. They present six programs in two versions each: with and without expression folding (i.e., inlining). They find that the same test suites performed very diversely on the two versions. Their suggestion is either (1) to introduce a new coverage metrics that takes the use of inlined condition into consideration, or (2) a canonical way of structuring code so that such conditions do not occur in the first place. Their approach differ from ours, since we propose a framework able to evaluate *existing* coverage criteria with respect to their robustness to code structure and syntax changes and to *extend* them by computing some information about their robustness/fragility. No change of the existing code is required in our approach either.

So far, the main solution in the literature to overcome coverage criteria weaknesses has been trying to introduce more powerful and tough testing criteria. For

instance, testing criteria that consider also the information flow can be introduced. In [18], the authors define a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. We believe that introducing complex coverage criteria may be avoided if code transformations are taken into account as proposed in our approach.

Testing criteria are very sensitive to code structure also when used for test generation. Often the structure of the code makes hard the generation of tests, i.e. it reduces its *testability*, especially when test generation is performed automatically. The automated generation of adequate test data can be impeded by properties of the code itself (for example, flags, side effects, and unstructured control flow). For this reason *testability transformations* are introduced [11]. A testability transformation is a code-to-code transformation that aims to improve the ability of a given test generation method to generate test data for the original program. A first difference between our approach and the work of Harman et al. is that we do not tackle the problem of test suite generation but we only want to measure the robustness of a given test suite in order to obtain a measurement of how much the coverage is affected by modifications in the code structure. Another difference is that *testability transformations* are not semantic preserving while the transformations defined in our work do not modify the semantics of the program. We will also show that the use of *testability transformations* should be carefully considered because the test suite generated from a transformed program P' that achieves a certain coverage C' may not achieve the same level of coverage C on the original program P.

Transformations and code coverage is studied by Weissleder [22]. In this case the transformation is used to obtain information of the coverage over the original code from the information about the coverage over the transformed code. The goal is to find a transformation such that if a test suite achieves the coverage C_1 over the transformed code, than the same test suite achieves the coverage C_2 over the original code. In this case C_1 *simulates* the coverage C_2.

The fact that transformations can disrupt coverage is also tackled by Kirner. In [12], he addresses the challenge of ensuring that the structural code coverage achieved for a program P is preserved when P is transformed. If the code transformation fulfills some formal properties, than it preserves also the coverage. The considered code transformations allow to obtain machine level code from higher level programs. He also identifies three classes of transformations: 1. the ones that change the reachability of program elements, 2. the ones that add new paths to the program and 3. the ones that preserve the coverage. His work is more focused on preserving structural code coverage into compilers and code generators, whereas our work is more focused on measuring the impact of transformations of the code structure on the coverage.

3 Theoretical background

Testing criteria, often called *coverage criteria*, have the main goal of measuring the test quality. They are used as a metrics to quantify the coverage of the control flow of a program achieved by a test suite. Usually they are also used as a stopping rule to decide when the testing process can stop and the software

can be accepted. Studying coverage criteria, defining new ones, and providing empirical evidence of their fault detection capability have been a major research focus for the last three decades. First of all, we introduce a framework defining them formally, mainly taken from [24], where the reader can find an exhaustive treatment of the subject. For the purpose of our paper we do not consider directly the use of testing criteria for test suite generation, and we focus on *program based structural testing*, that does not consider the specification and defines testing requirements only in terms of program elements (statements, conditions, and so on). Given the set of programs P and the set of all the test suites TS, we define a testing criterion in the following way.

Definition 1. ***Testing Criteria****. A testing criterion is a function C,* $C : P \times TS \rightarrow [0, 1]$*.* $C(p, ts) = r$ *means that the adequacy of testing the program* p *by the test set* ts *is of degree* r *according to the criterion* C*. The greater the real number* r*, the more adequate the test suite* ts *is.*

Given a fixed value r', such that $0 \leq r' \leq 1$, which represents the lower expected coverage for the criterion C applied to the program $p \in P$, we can consider the test suite $ts \in TS$ as adequate for testing program p iff $C(p, ts) \geq r'$.

In program based structural testing, coverage requirements are expressed in terms of the coverage achieved over a particular set of elements in the structure of the program under test (e.g. the set of all the statements for *statement coverage* or the set of all the conditions for *condition coverage*). We will focus on classical coverage criteria [3,15], including the statement and branch coverage and the Modified Condition Decision Coverage (MCDC) [5].

3.1 Code transformations

There exist several, theoretically infinite, programs which have the same behaviour but have different code structure and thus can achieve different results in terms of coverage for a particular criterion. So how, given a program $p \in P$, can we obtain new programs with the same behaviour of p? We can do this by means of *code-to-code transformations*, which are functions that take as input a program p and return another program p'. Formally, a transformation t is a function $P \longrightarrow P$, where P is the set of all the programs.

However not all the transformations produce a transformed program with the same behaviour of the original program. This kind of transformations are called Semantic Preserving Transformations (SPT). A SPT [2] is a code-to-code transformation that modifies the syntax of the program to which it is applied, without changing its semantics. Thus given a SPT $t \in T$, where T is the set of all the SPTs, and a program $p \in P$, p and $t(p)$ must have the same behaviour. In the following of the paper we will consider only SPTs and, thus, we will call them just transformations.

In the following we present five SPTs which we will use in the paper as a case study. Each transformation is identified by using a transformation schema, composed by two snippets of code. The first one, called input pattern, defines on which snippets of code the transformation can be applied. The second one, called output pattern, defines how the transformed piece of code will look like.

Several new transformations can be obtained by combining already defined transformations. Given a sequence of transformations T, we define the transformation t_{seqT} as the application of the transformations in T in sequence, i.e. $t_{seqT} = t_1 \circ \ldots \circ t_n$ where $t_{seqT}, t_1, \ldots, t_n \in T$. Given a transformation t, we define the transformation $\tilde{t}$ as the iterative application of t until the program is no longer modified by t. Given a certain transformation t, we can define the inverse transformation t^{-1} by exchanging the input pattern and the output pattern.

In this paper, we consider the following transformations.

Externalized Complex Flag. This transformation was already identified by Rajan et al. [17], which showed its effects on MCDC criterion, and by Harman et al. [11], which showed how it can be used in order to enhance the test generation phase. It has the following schema:

```
boolean x;                                  boolean x;
...                                         ...
x = complexBoolExpr; //A        t_ecf       x = complexBoolExpr;
...                              ⇒          ...
if (...x...){//B                            if (...complexBoolExpr...){
   ...                                         ...
}                                           }
```

In the schema, complexBoolExpr is a Boolean expression that contains at least one Boolean operator, and the statements between the point A and B do not change the value of x, and of the variables referenced in complexBoolExpr. We briefly call this transformation t_{ecf}. By applying $\tilde{t_{\mathsf{ecf}}}$ to a program p we obtain a new program $\tilde{t_{\mathsf{ecf}}}(p)$ in which all the flags in *if statements* are expanded to their definition. Several refactoring patterns [8] can be partially mapped on this transformation or its inverse, i.e. *Inline Temp Variable* (in case the variable is boolean and it is inlined in an *if statement*), *Remove Control Flag*, *Introduce Explaining Variable*.

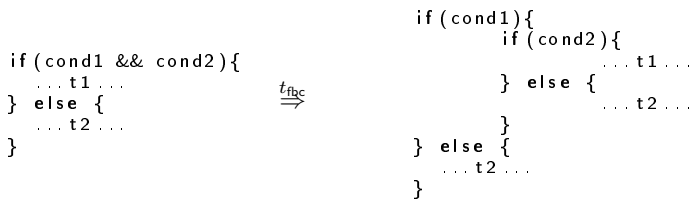
Boundary extraction This transformation t_{b} acts on an *if statement* and splits it into several *if statements* if it contains a condition in the form of $a \lessgtr x \lessgtr b$, where a, b, and x are numerical constants or variables. It has the following schema:

```
...t0...                                    ...t0...
if(a<=x && x<=b){                           if(x==a) {
   ...t1...                                    ...t1...
} else {                        t_b         } else if(x==b) {
   ...t2...                      ⇒             ...t1...
}                                           } else if(x>a && x<b) {
                                               ...t1...
                                            } else {
                                               ...t2...
                                            }
```

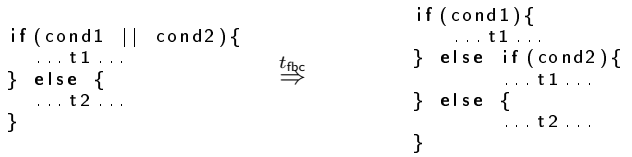
Reverse Conditional The transformation t_{rc} is associated with the *Reverse Conditional* refactoring pattern [1]. It simply inverts the condition of the *if statement* and exchange the *then block* and the *else block* between them, and *cond* is a boolean expression. It has the following schema:

```
if( cond ){                                 if(! cond) {
   ...t1...                     t_rc            ...t2...
} else {                         ⇒          } else{
   ...t2...                                    ...t1...
}                                           }
```

Flattening Conditional Expression The transformation t_{fbc} splits all the expressions used as guards in *conditional statements* until every *if statement* has only an atomic Boolean expression as a guard. It can be defined using two schema. The first schema represents how the transformation splits a conjunctive condition:

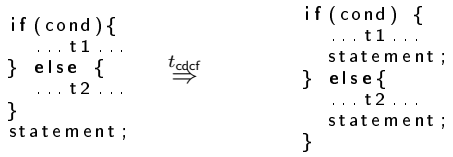


The second schema represents how the transformation splits a disjunctive condition:



In both schema *cond*1 and *cond*2 are boolean expressions. It is a generalization of the *Consolidate Conditional Expression* refactoring pattern [8]. Note that a similar transformation may be performed during compilation (e.g. in the byte code) and therefore, a test suite that achieves the decision coverage of the original program, may not achieve the same coverage of the compiled program. This problem is also studied in [12] and it is a common transformation done when the source code is transformed into assembly code (for conjunctive expressions).

Remove Consolidate Conditional Fragment The transformation t_{cdcf} is the inverse transformation of the one associated with the *Consolidate Conditional Fragment* refactoring pattern [8]. It simply moves into the *then block* and the *else block* the first statement after the *if statement*. It has the following schema:



Meaning of transformations The meaning of code transformations regarding the testing activity depends on which of two scenarios explained in the introduction we assume. If we suspect that the code was transformed in the past, code transformations bring the code to its original structure, while, if we assume that the code will be changed in the future, code transformations model the changes the code will be subject to. For instance, $\tilde{t_{\mathsf{ecf}}}$ would undo the insertion of flags for conditions in *if statements* done in the past.

4 Coverage robustness

4.1 Code transformations and coverage

Code coverage is very sensitive to the code structure and it can therefore be misleading as test adequacy criterion. This fact is well explained in [17] for MCDC in case of inlining and outlining of Boolean variables (addressed by our t_{ecf} and ${t_{\mathsf{ecf}}}^{-1}$ transformations). Even though examples in literature are focused only on the extrapolation of a complex flag, the sensitivity of coverage criteria to code structure can be generalized for any code transformation and any code coverage criterion. Indeed we show several code fragments with the same semantics that achieve different coverage degrees with the same coverage criteria and the same test sets.

Example 1. For instance consider the following code fragment, in which `in_1` and `in_2` are two inputs and the guard of the conditional statement has been *outlined*, i.e. the guard is simply a Boolean variable defined in terms of other Boolean variables or expressions.

```
boolean expr_1 = in_1 || in_2;
if (expr_1){                          ⇒ (t_ecf)      if (in_1 || in_2){
        ...                                                  ...
}                                                    }
```

A test suite containing only two tests (in_1=`true`, in_2=**false**) and (in_1=**false**, in_2=**false**) covers the MCDC for the *if statement*, which has a simple variable as guard, so two tests are enough. However, if we apply t_{ecf}, the transformed code were written with the condition *inlined* and thus the same test suite would not achieve the MCDC of the same code.

These simple patterns recur quite often in software and in models [17]. However, it would be not acceptable to force the developer to choose only the inlined version, in order to avoid that a full MCDC coverage is achieved with less test cases. The outlined version is more readable and maintainable since a complex expression is re-factored in an auxiliary variable. This situation can be also the result of an explicit *extract local variable* refactoring operation [8]. It could also perform better, since the Boolean flag is computed only once.

Example 2. Consider the following code, where x is an integer variable and a simplified form of boundary extraction is applied:

```
...
if (x>=2) x=x+1;      ⇒ (t_b)      ...
else      x=x+2;                   if (x==2)      x=x+1;
                                   else if (x>2)  x=x+1;
                                   else           x=x+2;
```

A test suite containing only two tests (x=**0**) and (x=**5**) achieves 100% of branch coverage. However the same test suite achieves only 75% of branch coverage on the transformed code.

Example 3. Consider the following code fragment, in which a and b are Boolean variables. To achieve a full decision coverage a test suite containing only two tests (a = **true**, b = **true**) and (a = **false**, b = **true**), is enough.

```
if(a && b){                          if(a) {
  .... // body      t_fbc ⇒            if (b){
}                                        .... // body
                                       }
                                     }
```

The transformation does not change the behaviour of the program, but the original test suite would cover only the first decision on the transformed program.

Example 4. Consider the following code fragment in which a is a Boolean variable and i an Integer variable:

```
if(a)                                if(a) {
      i = i+1;                             i = i+1;
else                                       System.out.println(i);
      i = i+2;              t_cdcf ⇒   } else {
System.out.println(i);                     i = i+2;
                                           System.out.println(i);
                                     }
```

The test suite containing only one test (a = **true**) achieves 60% of statement coverage. However on the code transformed by using the t_{cdcf} transformation, the same test suite achieves only 50% of coverage.

All the examples show that several programs with the same behaviour can achieve the same value of coverage with different effort from a testing point of view, i.e. the number of test cases in the test suite, only because they have a different structure. This is valid for all the structural coverage criteria.

4.2 Coverage fragility and robustness

First of all, we want to formalize the sensitivity of the coverage obtained by testing a program P with a test suite, with respect to a set of possible transformations of P.

Definition 2. ***Fragility****. Given a program $p \in P$, a coverage criterion C and a set of transformations T, we say that a test suite ts* fragilely covers *p, if there exists a transformation $t \in T$ such that $C(p, ts) > C(t(p), ts)$.*

Fragilely covered programs can be modified by some transformation $t \in T$ in a way that, also if the behaviour of the program remains the same, the coverage provided by ts on the transformed program $t(p)$ diminishes with respect to the coverage on the original program p.

If a test suite fragilely covers the program under test, the confidence in the measurement of the coverage is reduced because the possible high level of coverage may be due to the structure of the code. It may happen that the developer has used *in the past* a particular pattern that has increased the coverage but if the code were written in another way then the test suite would be not as good in terms of achieved coverage. Fragile coverage is not robust to transformations of the code that may be performed *in the future* either, such as refactoring techniques or compiler optimizations. This is a problem because, usually, after a SPT is applied, the test suite is not updated by the developer because he/she does not feel the need of new tests, and thus the old test suite can achieve lower coverage on the resulting code. For this reason, fragilely covered programs may need more testing, regardless the level of coverage achieved so far.

In order to reduce the fragility of a test suite, new tests must be added. Generally, the new tests are built looking at the transformed program and then added to the original test suite. However, using the transformed program to derive a completely new test suite to be applied also to the original program can cause an unexpected loss of coverage, as proved by the following theorem.

Theorem 1. *Let C be a coverage criterion, ts and ts' be two test suites, generated respectively for p and for $t(p)$, such that $ts \not\subseteq ts'$, $ts' \not\subseteq ts$, and $t \in T$ be a transformation, $C(t(p), ts') > C(t(p), ts)$ does not imply $C(p, ts') > C(p, ts)$.*

Proof. We prove the theorem by showing a case in which the converse – $C(t(p), ts') > C(t(p), ts)$ implies $C(p, ts') > C(p, ts)$ – is false. Consider the transformation t_{ecf} and the following program p and its transformed version $t_{\mathsf{ecf}}(p)$:

```
x = a && b && c;                    x = a && b && c;
if(x) {                  t_ecf      if(a && b && c) {
  ...                    ⇒            ...
}                                   }
```

Given the test suite ts, which has two test cases ($a = \mathbf{true}, b = \mathbf{true}, c = \mathbf{false}$) and ($a = \mathbf{true}, b = \mathbf{true}, c = \mathbf{true}$), and considering the condition coverage criterion, the coverage is $C(p, ts) = 1.0$ while $C(t_{\mathsf{ecf}}(p), ts) = 4/6 = 0.66$. If we consider then a test suite ts' that has two test cases ($a = \mathbf{false}, b = \mathbf{true}, c = \mathbf{true}$) and ($a = \mathbf{true}, b = \mathbf{false}, c = \mathbf{true}$), the coverage that ts' achieves on $t_{\mathsf{ecf}}(p)$ is $5/6 = 0.83$ and thus it improves the coverage of ts over $t_{\mathsf{ecf}}(p)$, i.e. $C(t_{\mathsf{ecf}}(p), ts') > C(t_{\mathsf{ecf}}(p), ts)$. Moreover, $ts \not\subseteq ts'$, $ts' \not\subseteq ts$ is true. However, the coverage of ts' over p dimishes, since $C(p, ts') = 0.5$ whereas $C(p, ts) = 1.0$.

Theorem 1 states that, given a program p fragilely covered by a test suite ts and a transformation t, if we want to achieve a better coverage than the one obtained using the test suite ts on the program p we cannot simply generate a new test suite ts' on the program $t(p)$, because ts', also if increases the coverage on $t(p)$ with respect to ts, maybe does not increase the coverage achieved on p.

Note that transformations allowing testers to obtain programs from which tests can be generated more easily are also called *testability transformations* [11]. Theorem 1 states that the use of testability transformations should be carefully considered since they may not increase the coverage obtained on the original program, even though the coverage is increased on the transformed program.

Sometimes we want to refer to the *coverage* as either fragile or robust. In accordance with Def. 2, we can introduce the following definition.

Definition 3. ***Fragile [Robust] Coverage****. Given a coverage criterion C, a program p, a test suite ts for p, and T a set of code transformations, we say that the coverage of p provided by ts with respect to the coverage criterion C and the code transformations T is* fragile *[*robust*], if and only if the program p is [is not] fragilely covered by ts with respect to C and T.*

The fact that a coverage is fragile or robust strongly depends on the set of transformations T one considers. With a small set T any coverage is likely to be robust, but with a large T only the best test suites will provide the robust

required coverage. For this reason, the client who requires certain levels of coverage and robustness has to provide the tester with adequate transformations in order to ensure the desired confidence in the code coverage. The given set of transformations T should depend on the expected set of SPTs that will be applied on the program or on the set of transformations applied in the past on the program. It is important to define such programs whose coverage is not affected by the application of transformations, we call them *robust* programs.

Definition 4. ***Robust program*** *Given a coverage criterion C and a set of code transformations T, we say that a program p has* robust *structure if* any *test suite that provides the coverage C for p, C is robust.*

Code with robust structure is of great interest for testers, since its coverage during testing cannot be diminished by code transformations, i.e. the coverage achieved by any test suite on a robust program will not diminish regardless the sequence of SPTs $t_1, \ldots, t_n \in T$ applied to it. Given a program p and a test suite ts, which achieves a certain level of coverage $C(p, ts)$ for a coverage criterion C, there are two ways to increase the achieved coverage. If the program p is robust, we can generate a new test suite ts' also transforming the program, because generating a test suite ts' that achieves a coverage higher than $C(p, ts)$ assure also that $C(p, ts') \geq C(p, ts)$. If the program is not robust the only way to increase the coverage is extending the test suite ts, because the Theorem 1 proves that generating a new test suite ts' that achieve higher coverage on a transformed version of p does not assure to obtain a better coverage on p.

4.3 Fragility and robustness measures

We define a measure to express how much the coverage achieved by the test suite ts on the program p with respect to the criterion C is robust to the changes in the code structure introduced by a set of SPTs T. Our metrics works with any existing coverage criterion C, without the need to introduce new and possible more complex testing criteria. This allows the tester to re-use existing criteria (and associated tools) which he/she is already familiar with.

The metrics is called *extended coverage* (because extends the usual coverage measurement with an information on how much the coverage offered by the test suite is sensitive to transformations), and it consists of a couple of values (a, b) where $a = C(p, ts)$ represent the usual coverage obtained by applying ts to the program p, whereas b is a fragility index such that $b \in [0, 1]$, and it measures the sensitivity of the coverage to modifications in the code structure. For a coverage measure in the form (a, b), $b = 0$ means that a is the robust coverage, while as $b \to 1$ the coverage a is increasingly sensitive to the transformations of the code structure. If $b = 1$ the coverage a is completely fragile.

Let p be a program, ts a test suite, and C a coverage, we define $\Delta(t) = C(p, ts) - C(t(p), ts)$ where t is a transformation. Let $pos(x)$ be a function defined as $max(0, x)$. We define three fragility indexes. The first one is simply the *averaged fragility*:

$$b_{af} = pos\left(\frac{\sum_{t \in T} \Delta(t)}{|T|}\right)$$

The second one is called *weighted fragility* and it is defined as

$$b_{wf} = \sum_{t \in T} \rho(t) * pos(\Delta(t))$$

where $\rho(t)$ is a function that defines the weight of each transformation $t \in T$, such that $\sum_{t \in T} \rho(t) = 1$. The *weighted fragility* is a useful metrics in case we want to assign a different weight to some transformations for a particular reason, e.g. the weight can represent the likelihood that a certain transformation will be applied to the code. The third fragility index is called *worst case loss of coverage*. It is an indicator of what is the maximum loss of coverage between the original program and any transformed one, and it is expressed as

$$b_{wc} = pos(max_{t \in T}(\Delta(t)))$$

In this case if the index $b_{wc} = 0$ it means that a is the robust coverage, otherwise b_{wc} indicates the maximum loss of coverage and thus the real coverage in the worst case is $C(p, ts) - b_{wc}$.

The *extended coverage* is a very useful metrics, especially if it is measured during the development phase. For instance with respect to unit testing, once the developer has measured the extended coverage, if it is not a robust coverage he/she can act in two ways to increase the robustness of the coverage: (a) he/she could extend the test suite with new test cases, maybe generated from a transformed version of the program, (b) he/she could change the structure of the code in order to remove all the points that introduces fragility issues. However, removing fragility points may be not straightforward nor possible every time (this fact highly depends on the transformations in T). Moreover transforming the code would increase the robustness at the expenses of the coverage, which would diminish. To maintain the same level of coverage, the tester should add new tests in any case.

5 Experiments

In order to evaluate how much the transformation of a program influences the robustness of the coverage offered by a test suite we have analyzed several Java programs. The selected programs vary from toy examples to complex Java libraries. The programs are the following:

LEAP: It contains one method checking whether the passed year is leap or not.

TRI: It contains a triangle classification method, which takes as input the length of the three sides and computes the type of the triangle [3].

WBS: The Wheel Brake System (WBS) is a Java implementation of the WBS case example found in ARP 4761 [4,17]. The WBS determines what pressure to apply to braking based on the environment.

TCAS: It is the Java implementation of a Traffic Collision Avoidance System (TCAS) II, required on all commercial aircraft flying in US airspace.

ASW: The Altitude Switch (ASW) is a synchronous reactive component from the avionics domain that controls the power of some devices according to the airplane altitude.

Program p	LEAP						TRI						TCAS					
LOC	10						35						155					
Coverage	Stmt		Branch		MCDC		Stmt		Branch		MCDC		Stmt		Branch		MCDC	
Test suite ts	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}	a	b_{wc}
Random1	100	0	100	0	100	50	93	0	94	0	95	5	65	15	23	13	18	8
Random2	100	0	100	0	100	50	93	0	94	0	95	5	-	-	-	-	-	-
Evosuite	100	0	100	0	100	13	100	0	100	0	100	0	94	22	87	43	74	30
Handmade	100	0	100	0	100	63	100	0	100	0	100	20	26	7	3	2	3	1

Table 1. Results in percentage of the robustness analysis for LEAP, TRI, and TCAS and their test suites, where $a = C(p, ts)$. Random1 test suite contains 100 tests, and Random2 contains 1000 tests for all projects. Evosuite has 4 tests for LEAP, 13 for TRI, and 16 for JTCAS. Handmade has 1 test for all the projects.

JTOPAS: It is a simple, easy-to-use Java library for the common problem of parsing arbitrary text data.

ANT: Apache Ant is a Java library and command-line tool widely used to compile and build Java applications.

NXML: NanoXML is a small XML parser for Java.

Code for TACAS, JTOPAS, ANT, NXML and their unit tests can be found in the SIR repository [7]. The Java implementation of ASW is included in the Java Path Finder distribution [21].

In our study we have considered statement, branch and MCDC coverage criteria. We have studied all the transformations presented in Section 3.1, and thus $T = \{\tilde{t_{\mathsf{ecf}}}, \tilde{t_{\mathsf{rc}}}, \tilde{t_{\mathsf{b}}}, \tilde{t_{\mathsf{fbc}}}, \tilde{t_{\mathsf{cdcf}}}\}$. For each example we have considered an handmade test suite (for TACAS, JTOPAS, ANT and NXML the considered test suite is the one presented in the SIR repository). For the smallest case studies (LEAP, TRI, WBS, and TCAS), we have also considered a test suite automatically generated by means of Evosuite [9] and random test suites with a fixed dimension of 100 and 1000 test cases. Experiments with random test suites are repeated 10 times, with different seeds, and only the averaged results are presented in this paper. For each test suite ts, for each coverage criterion C and for each program under test p, we have computed the coverage[1] achieved by ts for the criterion C on the program p and for any transformed program $t(p)$, for each $t \in T$.

Table 1 presents the results of the robustness analysis for the smallest case studies (LEAP, TRI, and TCAS). For each program and test suite, the table shows the results in terms of coverage a and the *worst case loss of coverage* b_{wc} fragility index.

Table 2 shows the results of our study on the biggest case studies (ASW, TOPAS, NXML, and ANT). For each program, the table shows the results in terms of coverage achieved by the provided test suite and the two fragility indexes, the *worst case loss of coverage* b_{wc} and *averaged fragility* b_{af}.

Table 3 shows the detailed results of the robustness analysis for the WBS program. For each test case and coverage criterion, the table shows the results

[1] We use the CodeCover tool `http://codecover.org/`

	ASW			TOPAS			NXML			ANT		
LOC/Classes	1497/47			10115/91			3696/34			104304/1266		
Test suite size (LOC)	965			4725			4231			24384		
Coverage	a	b_{wc}	b_{af}	a	b_{wc}	b_{af}	a	b_{wc}	b_{af}	a	b_{wc}	b_{af}
Statement	32.5	0.6	0.1	78.5	30.4	10.2	10.6	3.2	0.7	10.9	8.4	1.8
Branch	27.8	1.0	0.2	69.8	28.3	7.5	4.1	2.1	0.5	7.6	5.7	1.2
MCDC	31.1	1.1	0.2	69.4	30.6	7.6	4.8	2.6	0.5	7.6	5.9	1.3

Table 2. Results in percentage of the robustness analysis for ASW, TOPAS, NXML, and ANT

		Statement Coverage				Branch Coverage				MCDC			
Test suite	size	C	Δ			C	Δ			C	Δ		
			$\tilde{t}_{\mathsf{cdcf}}$	$\tilde{t}_{\mathsf{ecf}}$	$\tilde{t}_{\mathsf{fbc}}$		$\tilde{t}_{\mathsf{cdcf}}$	$\tilde{t}_{\mathsf{ecf}}$	$\tilde{t}_{\mathsf{fbc}}$		$\tilde{t}_{\mathsf{cdcf}}$	$\tilde{t}_{\mathsf{ecf}}$	$\tilde{t}_{\mathsf{fbc}}$
Random1	100	58.1	34.9	0	17.7	51.4	33.7	0	19.1	50.0	31.8	4.5	17.7
Random2	1000	74.2	56.1	0	22.5	74.3	59.7	0	28.3	71.2	56.4	10.3	25.2
Evosuite	6	74.2	56.1	0	22.5	74.3	59.7	0	28.3	71.2	56.4	10.3	25.2
Handmade	1	58.1	42.4	0	17.7	50.0	39.5	0	20.2	46.3	35.8	8.1	16.5

Table 3. Results in percentage of the robustness analysis for WBS (194 lines of code)

in terms of coverage achieved by the test suite and the losses in terms of coverage on the transformed versions of the program. The table shows only the results for the t_{fbc}, t_{ecf} and t_{cdcf} transformations, because the WBS program is not influenced by the t_{rc} and t_{b} transformations.

All the programs and test suites considered in our study suffer from fragility problems: semantically equivalent programs achieve different results in terms of coverage, and thus SPTs can influence greatly the coverage achieved by test suites. Our results highlight the fact that some transformations influence only certain coverage criteria, e.g. t_{ecf} influences only MCDC in our study, whereas other transformations, such as t_{rc}, seems to not influence the coverage at all. For this reason, the choice of the transformations considered in the robustness analysis can significantly influence the results of the analysis itself.

No apparent correlation can be identified between the size of the test suite and their fragility: indeed, significant losses in terms of coverage exist also for big test suites, e.g. Table 3 shows that also the random test suite with 1000 test cases has high losses. Note that even if the losses in terms of coverage may be small in some cases, this is usually due to the low coverage achieved by the test suites. For instance, in the ANT case the maximum loss is 8.4%, but the coverage achieved on the original program is only 10.9%.

Test suite with high coverage, can be fragile as well: from the results, it seems that test suites generated by hand explicitly to achieve good coverage, are those with higher losses in terms of coverage. This is due both to the fact that these test suites have small sizes and also to the fact that they are created ad-hoc to obtain full coverage of the program with a particular structure and thus the coverage is more fragile than the one of a not ad-hoc test suite.

6 Threats to Validity

There are three main aspects that can pose a threat to the validity of our work.

Transformations: Although the set of selected transformations is small, in our opinion it can demonstrate the effectiveness of our approach. By extending the given set of transformations, test suites become likely less robust. The selected transformations are meaningful examples. Indeed t_{ecf} is already used in several works [11,17], whereas t_{cdcf}, t_{rc}, t_{ecf} and t_{fbc} are extracted from common refactoring techniques [8,1].

Coverage Criteria: We have considered three common structural coverage criteria, i.e. statement coverage, branch coverage and MCDC. Rajan et al. [17] show that MCDC is highly sensitive to the structure of the implementation and our experiments confirm that. Test suites adequate to other non structural coverage criteria may be less fragile.

Experiments: Our work has focused only on a limited set of Java programs. However we think that chosen programs are representative of several classes of systems, i.e. toy examples (LEAP, TRI), critical systems (WBS, ASW, TCAS), and complex Java libraries (JTOPAS, NXML, ANT). Our experiments involved 1442 classes and more than 120kLOC, and therefore the selected programs are, in our opinion, a representative sample of real Java programs. We have also used different test suites which range from manually built test suites, to test suites generated by using well-known tools.

7 Conclusions and Future Work

In this paper we have proposed a framework to evaluate the robustness of a test suite with respect to semantic preserving transformations applied to the program under test. We have introduced the concept of fragile and robust coverage and we have identified the conditions for a code to have a robust structure with respect to a certain set of transformations. Moreover, we have defined a new extended coverage metrics that takes into account the fragility of the coverage. The extended coverage does not require either a modification of the code or the introduction of new original testing criteria. It uses a fragility index to quantitatively measure the quality of test suite in terms of its robustness. In presence of fragile code, we suggest either to (1) find and remove fragility points by modifying the code or (2) increase the test suite until its robustness reaches a desired level. We have evaluated the fragility of several Java programs (from toy examples to Java library code) together with their test suites and we have found that the fragility problem occurs in all the considered programs.

In the future we plan to study the correlation between the fragility of the test suite and its fault detection capability. We also plan to define a language for the formalization and definition of semantic preserving transformations. In this way, we can easily model other transformations and also extend the theoretical framework.

Acknowledgments The authors would like to thank Matt Staats for sharing some code examples.

References

1. Refactoring catalog - website. `http://www.refactoring.com/catalog/`.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.
4. *ARP 4761,Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Aerospace Recommended Practice, Society of Automotive Engineers, Detroit, USA, 1996.
5. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.
6. E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.
7. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
8. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.
9. G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of ACM SIGSOFT ESEC/FSE*, pages 416–419, 2011.
10. J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Softw. Eng.*, 1(2):156–173, June 1975.
11. M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Softw. Eng.*, 30:3–16, January 2004.
12. R. Kirner. Towards preserving model coverage and structural code coverage. *EURASIP J. Emb. Sys*, 2009:1–16, 2009.
13. B. Marick, J. Smith, and M. Jones. How to misuse code coverage. In *Proc. of ICTCS'99*, June 1999.
14. J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6:58–63, February 1963.
15. G. J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.
16. A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. of ISSTA 2009*, pages 57–68. ACM, 2009.
17. A. Rajan, M. W. Whalen, and M. P. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Proc. of ICSE*, pages 161–170, 2008.
18. S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Soft. Eng.*, SE-11(4):367 – 375, april 1985.
19. M. E. Senko. A control system for logical block diagnosis with data loading. *Commun. ACM*, 3:236–240, Apr. 1960.
20. M. Staats, G. Gay, M. W. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *Proc. of Fundamental Approaches to Soft. Eng. (FASE)*, 2012.
21. M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proc. of ISSTA*, pages 183–194, New York, NY, USA, 2010. ACM.
22. S. Weißleder. Simulated satisfaction of coverage criteria on uml state machines. In *Proc. of ICST 2010*, pages 117–126. IEEE Computer Society, 2010.
23. E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM J. Comput.*, 8(4):587–598, 1979.
24. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.