Boolean Expression Extender - A Mutation Operator for Strengthening and Weakening Boolean Expressions

Simón Gutiérrez Brida*, Gastón Scilingo*
*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina

***Abstract*—The degree in which a software system is guaranteed to correctly provide its intended functionality, is one of the most relevant properties in the development of quality software, and the concern of software verification. Among the many techniques that approach software verification, testing, i.e., contrasting actual software behavior against expected behavior in a set of specific scenarios called test cases, is without any doubt the most widely used. Performing software testing appropriately demands evaluating the quality of the test cases, and mutation testing is a particularly relevant technique for evaluating the adequacy of test suites, that occupies us in this paper. Mutation testing proposes evaluating how good a set of test cases is by injecting artificial bugs in a program, and checking whether the suite is able to catch these buggy programs, called mutants. Mutation testing's performance is strongly related to the mutation operators considered, i.e., the set of changes that are applied to a program to generate mutants. We present a novel mutation operator whose main focus is boolean expressions, and that operates by strengthening and weakening such expressions. Although operators that perform this task already exist in the context of mutation testing, e.g. the `COR` operator, these do not provide finer-grained changes in expressions. The rationale behind our new operator is that performing finer-grained mutations in boolean expressions results in more "stubborn" mutants, i.e., more mutants that are difficult to kill. We present our novel mutation operator in detail, discuss its characteristics and rationale, and evaluate the impact of introducing this operator in mutation testing tools, with respect to how it performs in assessing test suite quality.**

***Keywords*—Software engineering, Testing, Mutation, Boolean expressions, Equivalent mutants, Stubborn mutants, Mutation operator**

I. Introduction

Guaranteeing that a software system fulfills the tasks it is supposed to do is among the most challenging problems in Software Engineering, and a primary concern in Software Engineering research [19], [20], [21]. It is in fact a driving concern in software quality assurance, and a task that consumes significant development resources [21]. With the increasingly dynamic nature of software, that continuously evolves to adapt to alterations and extensions in requirements, the introduction of new software features, etc., ensuring that software behaves as expected becomes even more difficult, since it includes checking, e.g., that new extensions do not break previous features, and other behavior preservation properties. Among the many techniques that approach the task of guaranteeing that software operates correctly, testing, i.e., contrasting actual software behavior against expected behavior in a set of specific scenarios called test cases, is without any doubt the most widely used.

Roughly speaking, testing a piece of code whose behavior one wants to assess simply consists of executing the code in a number of specific situations, and evaluating how the software behaves [13]. Of course, exhaustively executing a program, i.e., under all its possible inputs, is generally infeasible, and thus a sample of all possible inputs must be taken for testing. This makes testing necessarily incomplete, but at the same time implies a scalability that other verification approaches, especially static checking ones, are unable to handle. Clearly, how the sample is taken greatly influences the ability of testing in catching bugs, and developers therefore require some mechanisms to analyze whether a given sample being used for testing is suitable or not.

There exist many approaches to decide how suitable a test suite is, typically categorized as black box (not based on the structure of code) and white box (that take into account code's structure). Mutation testing is a particularly relevant technique for evaluating the adequacy of test suites, that may be considered white box, and that occupies us in this paper. In essence, mutation testing proposes evaluating how good a set of test cases is by injecting artificial bugs in a program, and checking whether the suite is able to catch these buggy programs, called *mutants*. Mutation testing's performance is strongly related to the mutation operators considered, i.e., the set of operators that produce the changes that are applied to a program when generating mutants. In this paper, we present a novel mutation operator, that we refer to as *Boolean Expression Extender* or `BEE`, whose main focus is boolean expressions. This operator works by strengthening and weakening such expressions, in ways that are not covered by existing mutation operators. Indeed, although operators that perform changes to boolean expressions already exist in the context of mutation testing, e.g. the `COR` operator, these do not produce in boolean expressions the changes of finer granularity that our operator produces. The rationale behind our new operator is that, by performing finer-grained mutations in boolean expressions, one obtains as a result a greater number of "stubborn" mutants, i.e., an increased number of mutants that are difficult to kill.

We present our novel mutation operator in detail, discuss its characteristics and rationale, and evaluate the impact of introducing this operator in mutation testing tools, with respect

978-1-5386-3057-0/17/$31.00

to how it performs in assessing test suite quality, on a number of case studies taken from an existing repository.

II. Background

Testing consists of executing a program in a number of specific (concrete) scenarios, and checking that the actual results of the executions match, in some way, the expected results. A test typically consists of an input for the program under test, the execution of the program (e.g., a call of a method whose behavior one wants to evaluate) and a set of assertions regarding the result of the execution, being it the return values and/or the state of the program after the execution finishes. A test suite is a set of tests usually grouped together by a same goal, like testing the same component of a program.

If we were to create a test suite to validate a particular algorithm, we could use every possible input and check that the corresponding output is correct. This is obviously infeasible and even if we could generate a big set of inputs we need our tests to finish in a reasonable amount of time. Thus we need a metric to evaluate how good a test suite is in discovering faults, so that we can make smaller test suites that are good enough at detecting faults but take less time and resources to run. Test suite evaluation approaches measure, in some way, the adequacy of a test suite to detect faults. Since one can only detect that a bug is present (by giving a test that exposes that bug), it is in principle not possible through testing to say that there are no bugs in a piece of software. Therefore, the ability of a suite in detecting faults is a metric that cannot be used. Indirect metrics are used instead. These metrics include code coverage metrics, that measure how well a test suite exercises a particular program's code. This test suite evaluation technique is based on the idea that if more code is exercised then more faults should be detected. Some code coverage criteria are: branch coverage, which evaluates how many branches a test suite executes; and statement coverage, that simply measures how many statements are executed by a given test set. These metrics can be used for test suite comparison: if a suite has a good coverage value then it should have a better capacity to detect faults than another test suite with a lower coverage value.

Mutation testing is a particular test suite evaluation technique, that concerns us in this paper. It is based on artificially seeded faults generated by making simple syntactic changes to an original code. Mutation testing works by taking a test suite and a program in which all test pass, and then produces versions of the original program in which a single syntactic change is made, each of which is called a *mutant* of the original code. Then, the test suite is executed on each mutant, checking whether tests pass or not; if at least one test fails then the corresponding mutant is marked as *killed*. A mutation score can be calculated for the test suite as a ratio between killed mutants and all mutants. This mutation score is a way to evaluate a test suite in which a higher value means the test suite is able to detect more faults. Killing a mutant means the test suite is able to detect the artificial fault. In [5] Ren Just et al. conclude that mutants are a valid substitute for 75% of the analyzed real faults.

Fig. 1: mutation example, Δ shows lines in which a single mutation has been applied

```
int median( int a, int b, int c ) {
  if (a <= b && b <= c || c <= b && b < a) {
  Δ(a > b && b <= c || c <= b && b < a)
  Δ(a <= c && b <= c || c <= b && b < a)
  Δ(a <= b && b <= c || true)
    return b;
  Δreturn b++;
  Δreturn a;
  Δreturn 0;
  } else if (b < a && a < c || c < a && a < b) {
    return a;
  } else if (a < c && c < b || b < c && c < a) {
  Δ(a < c || c < b || b < c && c < a)
    return c;
  }
  return a;
}
```

A mutation testing tool implements a series of mutation operators; each of these operators can be applied on specific elements in the program's code (mutation spots), e.g. binary expressions with a conditional operator, and generate changes called mutations, e.g., changing the operator in a boolean expression. Example of several types of mutation are shown in Figure 1.

III. Boolean Expression Extender

A. Motivation

How many operators are used by a mutation testing tool not only affects the cost of running the analysis (more operators lead to more mutants and then the analysis takes more time), but it also affects the confidence in the mutation score obtained. To understand this we need to define equivalent, easy to kill, and stubborn (or hard to kill) mutants. From [4] we can define an equivalent mutant as a variant of the original program in which the mutation did not change the semantics of the program (an example of an equivalent mutant is shown in Figure 2); a hard to kill mutant (or stubborn) can be defined as a mutant that can be killed (there is a specific test that can detect the introduced defect), but that test is not present in a thoroughly enough test suite; and an easy to kill mutant is a mutant that does not enter in the two previous definitions, and a test that can detect the introduced fault is easy to find; for example, a mutant that does not compile, or a mutant that change the value of a variable to `null` right before accessing a field of that variable, are easy to kill mutants.

Now we can see how these types of mutants affect the confidence of the obtained mutation score in the following ways: equivalent mutants decrease the mutation score (since they cannot be killed, by definition), but they are not introducing any faults to the test suite to detect; easy to kill mutants increase the mutation score but since the faults associated with these mutants are very easy to detect (sometimes they can be detected directly by the compiler) the increased mutation score

Fig. 2: equivalent mutations examples

```
int max(int a, int b) {
  if (a < b) {
  Δ(a <= b)
    return b;
  } else if (a > b) {
  Δ (a >= b)
    return a;
  }
  return a;
  Δ return b;
}
```

Fig. 3: example of mutation operators that weaken or strengthen boolean expressions

$$A \,\&\&\, B \rightarrow A \,\&\&\, true$$
$$A \,\&\&\, B \rightarrow true \,\&\&\, B$$
$$A \,\&\&\, B \rightarrow A \,||\, B$$
$$A \,\&\&\, B \rightarrow true$$
$$A \,\&\&\, B \rightarrow false$$

does not mean that the test suite can detect more faults; finally, hard to kill mutants should at first decrease the mutation score, since by definition the tests that can detect the fault associated to a stubborn mutant are not in the test suite, but having more hard to kill mutants means that if we start to expand a test suite then this new test suite can detect faults that are not easy to detect.

In this work we focus in mutation operators whose syntactic changes can be semantically viewed as weakening or strengthening boolean expressions. We can see in Figure 3 a few examples of mutation operators that, syntactically, change conditional binary operators and expressions by the constants `true` and `false`, but semantically, the effect they cause is the weakening or strengthening of boolean expressions.

B. The Boolean Expression Extender Operator

We propose a mutation operator that allows one to make more granular changes to boolean expressions, taking as a main approach the extension of existing expressions using other expressions found in the class being mutated. This new operator is called **BEE** as an acronym for `Boolean Expression Extender`.

Our operator can be described in a very simple way. Given `A`, `B`, and `E` boolean expressions, `op`, and `op´` binary boolean operators (logical and, or; bitwise and, or; and xor), our operator works as follows:

$$\begin{aligned} BEE(A) &= (A \;\; op' \;\; E) \\ BEE(A \;\; op \;\; B) &= (A \;\; op \;\; B \;\; op' \;\; E) \\ &= BEE(A) \;\; op \;\; B \\ &= A \;\; op \;\; BEE(B) \end{aligned}$$

Fig. 4: median example code

```
int median( int a, int b, int c ) {
  if (a <= b && b <= c || c <= b && b < a) {
    return b;
  } else if (b < a && a < c || c < a && a < b) {
    return a;
  } else if (a < c && c < b || b < c && c < a) {
    return c;
  }
  return a;
}
```

Fig. 5: median example BEE mutations

```
int median( int a, int b, int c ) {
  if ((a <= b && this.bfield) && b <= c || c <= b &&
       b < a) {
    return b;
  } else if (b < a && (a < c || bar()) || (c < a ||
      true) && a < b) {
    return a;
  } else if (a < c && c < b || (b < c && c < a &&
      foo())) {
    return c;
  }
  return a;
}
```

where `E` is a reachable boolean variable, a boolean class field, a boolean class method with no arguments, and the `true` and `false` constants. To avoid some equivalent mutants, we will not extend a boolean expression with `|| false` and `&& true`.

In Figure 4 we show the original code for a median function, and in Figure 5 we illustrate several `BEE` mutations of the original code. While we are showing several mutations at once it is important to remark that each mutant we generate will have, as usual, only one mutation.

IV. Evaluating the BEE Mutation Operator

Our methodology for evaluating our new mutation operator will be driven by the following research questions.

> *Does our new operator generate more stubborn mutants than existing operators that weaken or strengthen boolean expressions?*

We want to confirm if more granular changes in boolean expressions lead to more stubborn mutants, or at least to fewer "easy to kill" mutants.

> *How does our operator affect the mutation analysis?*

Although stubborn mutants are a good thing to get from a mutation operator, we want to verify what is the overall impact of our new mutation operator, by looking at: *(i)* how many mutants it adds to the analysis; *(ii)* how many of these mutants fail to compile; *(iii)* how many equivalent mutants our new operator generates (we do not want an operator that generates too many equivalent mutants); and *(iv)* for the killed mutants, how difficult it is to kill them.

In summary, we want to verify if complementing the mutation operators that weaken or strengthen boolean expression with our new operator that makes more granular changes results in more stubborn mutants (or fewer easy to kill mutants). In order to answer the above research questions, we will need to fulfill the following tasks.

- Implement our new operator.
- Obtain programs to which we will apply our mutation tool to generate mutants.
- Create reasonably thorough test suites for each program.
- Select a set of operators to use.
- Generate sets of mutants (with and without our new operator).
- Run a mutation testing tool with both sets of mutants for each program and their corresponding test suites, and analyze the surviving mutants to filter non-equivalent ones.
- Analyze the overall impact our new operator had over the mutation analysis.

A. Implementation of the new operator

We have been working on an extension of `muJava` [3] called `muJava++` [7], for a few years. Our new operator will then be implemented in this tool, since our extension simplifies how operators are written and provide various auxiliary features, such as obtaining all reachable variables from a specific point in the code, that enable a more straightforward implementation of our mutation operator.

B. Obtaining case study programs

After reviewing the literature, looking for programs used in evaluations such as the one we attempt in this paper, we finally decided to use the `SIR` repository [6]. Since mutation evaluation is known to be extremely costly, we are unable to evaluate our operator on all cases of the repository. We selected five of them, as follows. We downloaded all java programs that had approximately 500 LOC or less. For each one of the downloaded programs we sorted them first in size (LOC) and after that we sorted them in how many mutation spots for operators that make changes in boolean expression each of them had. We then took the first five programs.

Trityp: a program that takes three integer values corresponding to the sides of a triangle, and decides whether the triangle is scalene, isosceles, equilateral, or is not a valid triangle. This program consist of only one method that resolves the triangle problem.

ArrayPartition: this program is part of the quicksort algorithm and resolves the problem of partitioning an array in two, selecting a pivot value and moving all values less or equal to the pivot to the left side of the array and all values greater than the pivot to the right side of the array. This program consist of two methods, one that takes an array and returns a modified array with the elements reordered around a selected pivot, and a second method that validates that the returned array is weakly sorted, that is, all elements on the left of the pivot are less or equal of the pivot and all elements right of the pivot are greater.

BinaryHeap: an implementation of a binary heap offering the following methods. Insert, inserts a new element in the heap; deleteMin, deletes the minimum element in the heap and returns its value; findMin, returns the minimum value in the heap; isEmpty, returns true iff the heap is empty; and isFull, returns true iff the heap is full.

OrdSet: an implementation of an ordered set of integers. This implementation offers the following methods. Union, returns the union of two ordered sets of integers which is also ordered; contains a value, searches the set for a specific value and returns true if found; contains an ordered set, searches for a specific subset and returns true if found; remove, removes a specific value and returns true if found; add, adds a specific element to the set.

TreeMap: an implementation of a tree map offering the following methods. containsKey and containsValue, which take a value and returns true if it exists as a key or as a value, respectively; get, which gets the value associated to a key; put, which takes a key and a value and stores the association between the key and the value in the map (if the key was already associated to a value then the old value is returned); remove, that takes a key and removes the key and the associated value if these exist; it will also return the associated value.

When looking for suitable case studies, we initially considered `Defects4J` [32]. This is in fact a very interesting benchmark of real java projects. The provided repository of programs includes a "fixed" version of the program; a "bugged" version, whose differences with the fixed version is only the patch needed to fix that particular bug; a set of tests which are composed of positive tests that pass on both the buggy and fixed versions, and negative tests that expose the particular bug in the buggy version but passes on the fixed version. The main relevance of this benchmark is that the programs are real, with developer written tests. But the size of the programs would have made our analysis too complex to present a new mutation operator for the first time. Using smaller programs allowed us to make a simpler analysis that could eventually lead to a more complex one. We need to see how our operator works with these smaller programs before deciding what to do next.

C. Creating reasonably thorough test suites

First we will need to define what a reasonably thorough test suite is with respect to mutation testing. According to Xiangjuan Yao et al. [4], a reasonably thorough test suite is a test suite which covers all branches. But since branch feasibility is undecidable, and in order to reduce bias in the created tests, we will use an automatic test generation tool to produce such suites.

Our methodology will be to use `EvoSuite` [8] to generate tests with branch coverage as the only coverage goal. `EvoSuite` is a tool to automatically generate test suites using evolutionary search and coverage criteria. The generated tests

are sequences of method calls that exercise the code under test followed by simple assertions. After automatically generating test cases using EvoSuite, we will add assert statements and if we consider it necessary, we will add more test cases, manually.

D. Operators to use

For our experiments we will select all basic method-level operators implemented in muJava++. These operators are a greater set that includes the sufficient mutation operators defined in [2], except for ABS (which changes an arithmetic expression into 0, a positive value, and a negative value) because it was not implemented in muJava and thus was not implemented in muJava++. Other operators that are not included in the sufficient mutation operators set are closely related to them except for the SOR operator. BEE could generate equivalent mutants to those generated by our set of basic operators. We will evaluate BEE along side these basic operators to observe the contribution of mutants that are only generated by BEE.

AODS	Deletes every occurrence of short-cut arithmetic operators.
AODU	Deletes every occurrence of unary arithmetic operators.
AOIS	Insert short-cut arithmetic operators on each variable and field.
AOIU	Insert unary arithmetic operator on each arithmetic expression.
AORB	Replace basic binary arithmetic operators with other binary arithmetic operators.
AORS	Replace short-cut arithmetic operators with other short-cut arithmetic operators.
AORU	Replace basic unary arithmetic operators with other unary arithmetic operators.
ASRS	Replace short-cut assignment operators with other short-cut operators of the same kind.
COD	Delete unary conditional operators.
COI	Insert unary conditional operators.
COR	Replace binary conditional operators with other binary conditional operators.
LOD	Delete unary logical operator.
LOI	Insert unary logical operator.
LOR	Replace binary logical operators with other binary logical operators.
ROR	Replace relational operators with other relational operators, and replace the entire predicate with true and false.
SOR	Replace shift operators with other shift operators.

Unfortunately we cannot replace any of these operators with BEE because there are several changes in boolean expressions, that weaken or strengthen the original, that can only be covered by BEE in more than one mutation generation. For example, `A && B ==> A || B`, a COR mutation, cannot be covered by only one BEE mutation; we should in fact do something as the following

$$A \,\&\&\, B \rightarrow (A \,\&\&\, false) \,\&\&\, B \tag{1}$$

$$(A \,\&\&\, false) \,\&\&\, B \hookrightarrow (A \,\&\&\, false) \,\&\&\, (B \,\&\&\, false) \tag{2}$$

$$(A \,\&\&\, false) \,\&\&\, (B \,\&\&\, false) \hookrightarrow ((A \&\& false) \,\&\&\, (B \,\&\&\, false) \,||\, A) \tag{3}$$

$$((A \,\&\&\, false) \,\&\&\, (B \,\&\&\, false) \,||\, A) \hookrightarrow ((A \,\&\&\, false) \,\&\&\, (B \,\&\&\, false) \,||\, (A \,||\, B)) \tag{4}$$

and even though these operators generate mutations of the form `A && B => A && true`, a ROR mutation that can be covered by a single BEE mutation `A && B => A && (B || true)`. The ROR operator generates several mutations that cannot be covered, namely, mutations that change a relational binary operator by another.

E. Mutants generation

Since we will be using muJava++ in this study, generating both sets of mutants and running a mutation testing analysis with them, is only a matter of writing two separate properties files (containing configuration parameters for the run) in which one will have all basic method-level mutation operators and the other will include our BEE operator. muJava++ will then be run twice per program, one with a properties file that includes the BEE operator, and one without it.

F. Mutation testing and stubborn mutants analysis

muJava++ is able to run a very detailed mutation analysis. Some information that we can obtain is mutation score with and without non-compiling mutants; surviving mutants, i.e., the mutants that we are interested in since all non-equivalent surviving mutants will be considered as hard to kill; and toughness analysis, that gives for each killed mutant a relation between how many tests the mutant survived and the total tests run, this is not the same as declaring a mutant as hard to kill but can give a measure of how difficult a mutant is to kill.

The steps we will follow to count stubborn mutants and to verify if our new operator generates more of them will be as follows.

- Run muJava++ mutation analysis twice, one with and one without the new operator.
- For each surviving mutant we will then make a manual analysis to check equivalence, since the programs we are using are relatively simple and small this is an analysis that can be done. Even though this manual validation raises a threat to the validity of our claims, it must be noticed that the equivalence problem is undecidable so any analysis must be manual or automatic based on heuristics.

- Before the manual analysis we will build a more robust test suite, for example a bounded exhaustive test suite, to quickly detect some of the stubborn mutants, if present. This will be done for two reasons. The first one is time: we may have a big number of surviving mutants (this depends on how many mutants are generated). And the fact that most test suites done by `EvoSuite` will be focused on public methods. The second reason is to diminish the chances of human error. One person analyzing 10-40 small to medium sized programs to detect if they are equivalent or not is an easily achievable task, although tedious and very demanding, but analyzing 400 - 1000 survivors (a value that can be achieved) will be prone to errors.
- Since `muJava++` also provides a toughness analysis that can be done at the same time it runs the mutation score analysis, we will also look at this value to see if the average toughness also increases when adding our new operator. We will use this value to see if `BEE` maintains the difficulty to kill mutants, decreases it, or the generated mutants are harder to kill. The relation of the values obtained by adding `BEE` to how many new mutants it added will be used to see how much impact did it have on the changes in the different results of the mutation analysis.

In [10], the author defines a stubborn mutant as one that can only be killed by less than `0,1%` of the inputs. This definition is very interesting and we could use it since `muJava++` gives the toughness of each mutant, how many tests failed to detect it, and we could define a stubborn mutant as one with a toughness value greater than `0,99`. But to do this analysis we would need to change how we generate the test suites. Nevertheless is an interesting point to look at in future work.

V. Experimental Results

After running `muJava++` for each program with and without `BEE` we collected the following values.

Mutant How many programs, each with one mutation, were generated by `muJava++`.

Killed How many mutants were detected when running the program's associated test suite.

Surviving How many mutants were not detected when running the program's associated test suite.

Failed to compile How many mutants were killed because they failed to compile.

Toughness How many tests in average are passed by the mutants. This is an indirect metric for measuring how hard it is to kill the mutants.

Toughness (killed) The same as toughness but this is only calculated on killed mutants.

Mutation score The ratio between killed mutants and total number of mutants.

Stubborn How many stubborn mutants of the surviving ones were detected by the manual analysis.

The results for the **Trityp** program can be seen in Figure 6. This was a case of a program that did not benefit much from `BEE`, in relation to stubborn mutants, since it does not have any boolean variables, fields, or methods, so the only possible mutants are the ones using the constants `true` and `false`. `BEE` generated **120** more mutants of which all compiled and **114** where killed by tests. The **6** surviving mutants were stubborn where the mutation caused that parts of some conditions to be removed (`Side1 <= 0 && false` for example). The test suite did not detect these faults because branch coverage does not need to cover every boolean combination of a condition, for `if (a || b) ...` a test where `a` is `true` and one where both are `false` covers both branches. `BEE` forces to cover more combinations for a condition. The overall impact of our new operator for this case was quite good. Mutation score and toughness values stayed almost the same. From the **120** mutants, **5%** where stubborn and no equivalent mutants were introduced. For this case we could replace some mutation operations with `BEE`, for example `ROR`'s mutation operation that changes a boolean condition to either `true` or `false`, could be removed from the operator and use `BEE` instead. That mutation operation in `ROR` should not be part of that operator in the first place, since it has nothing to do with `Relational Operator Replacement`.

The results for the **ArrayPartition** program can be found in Figure 7. This program also did not benefit much from `BEE`, in relation to stubborn mutants. In this case the problem was that the `partition` method generated many equivalent mutants where `isWeaklySorted` was used in a condition, but at the same time `isWeaklySorted` is a method that defines the postcondition of `partition`. There was no other variable, method, or field to be used. `BEE` generated **150** more mutants of which all compiled and **288** were killed by tests, from the **22** surviving mutants, **2** were stubborn. The stubborn mutants were caused by extending a condition with `isWeaklySorted`. The mutants needed a specific test with a non weakly sorted array. The overall impact of our new operator for this case shows little improvement. Mutation score and toughness values stayed almost the same. From the **150** mutants, only **1,33%** were stubborn but **13,33%** were equivalent mutants. We can see that `BEE` benefits from having several fields, methods and variables to choose from, and that it does not performs too well otherwise. This presents a question about complexity vs efficiency. With just variables, methods, and fields (of the same class) we depend too much on the existence of these elements in the same class we are analyzing. An option would be to include inherited methods and fields; to include boolean methods that have parameters, currently we only use methods without any parameter; boolean expressions found in the code, like comparisons and method calls. But this will also increase the complexity of the operator, giving more mutants and increasing mutation analysis cost.

The results for the **OrdSet** program can be found in

Fig. 6: Results table for Trityp program showing mutants, surviving mutants, mutants that failed to compile, toughness, mutation score, and stubborn mutants.

Program	No BEE	BEE
Mutants	435	555
Killed	326	440
Surviving	109	115
Failed to compile	0	0
Toughness	0,98	0,98
Toughness (killed)	0,98	0,98
Mutation score	74,94%	79,27%
Stubborn	56	62

Figure 9. For this program `BEE` almost doubled the amount of mutants. `BEE` generated **970** more mutants of which all compiled and **529** were killed by tests, from the **441** surviving mutants, **243** were stubborn. This is an interesting case, there are not many conditions using **or** so mutations where some parts of the condition are disabled can not be produced. However, since the `OrdSet` class provides several boolean methods and fields, `BEE` can generate mutations where a condition is changed but in some particular cases, for example using the `isEmpty()` method, where the condition will change only in some cases, in this case it will be affected by the set being empty or not. The overall impact of our new operator for this case was significant. Mutation score decreased by almost **15%** and toughness values stayed almost the same. This decrease in mutation score is clearly related to the increase in surviving mutants. From the **970** mutants, only **25%** were stubborn but **20%** were equivalent mutants.

The results for the **TreeMap** program can be found in Figure 10. In this case we discovered a large number of stubborn mutants. The main reason is because when we created the test suite using `EvoSuite`, it generated tests for branch coverage, but only for accessible methods, that is, public methods. This had as a consequence that private methods were not fully tested by the test suite. `BEE` almost quintupled the amount of mutants. More than half of the new mutants survived, and the mutation score dropped from **72%** to **45%**. This drop can be either because a lot of equivalent mutants were generated or a lot of stubborn ones were. `BEE` generated **1960** mutants of which all compiled and **744** were killed by tests, from the **1216** surviving mutants, **517** were stubborn. This was a difficult case to analyze since a complex structure must be created to be able to fully test its methods. From the **1960** mutants, only **26%** were stubborn but **35%** were equivalent mutants.

The results for the **BinaryHeap** program can be found in Figure 8. `BEE` generated **435** mutants of which all compiled and **205** were killed by tests, from the **230** surviving mutants, **98** were stubborn. From the **435** mutants, only **22%** were stubborn and **30%** were equivalent mutants.

Fig. 7: Results table for ArrayPartition, program showing mutants, surviving mutants, mutants that failed to compile, toughness, mutation score, and stubborn mutants.

Program	No BEE	BEE
Mutants	196	336
Killed	170	288
Surviving	14	36
Failed to compile	12	12
Toughness	0,54	0,58
Toughness (killed)	0,50	0,53
Mutation score	92,85%	89,28%
Stubborn	6	8

Fig. 8: Results table for BinaryHeap, program showing mutants, surviving mutants, mutants that failed to compile, toughness, mutation score, and stubborn mutants.

Program	No BEE	BEE
Mutants	454-	889
Killed	328	533
Surviving	111	341
Failed to compile	15	15
Toughness	0,79	0,83
Toughness (killed)	0,73	0,72
Mutation score	75,55%	61,64%
Stubborn	65	163

Fig. 9: Results table for OrdSet, program showing mutants, surviving mutants, mutants that failed to compile, toughness, mutation score, and stubborn mutants.

Program	No BEE	BEE
Mutants	1278	2248
Killed	1021	1550
Surviving	179	620
Failed to compile	78	78
Toughness	0,86	0,86
Toughness (killed)	0,83	0,83
Mutation score	85,99%	72,41%
Stubborn	160	403

Fig. 10: Results table for TreeMap, program showing mutants, surviving mutants, mutants that failed to compile, toughness, mutation score, and stubborn mutants.

Program	No BEE	BEE
Mutants	576	2536
Killed	409	1153
Surviving	160	1376
Failed to compile	7	7
Toughness	0,95	0,97
Toughness (killed)	0,93	0,94
Mutation score	72,22%	45,74%
Stubborn	114	631

Fig. 11: example for improved BEE boolean expression search

```
int foo(int a, int b, boolean retNan) {
  if (a > b) return a - b;
  if (a < b) return b - a;
  if (a == b) return 0;
  if (retNan) return NaN;
  else return -1;
}
```

VI. Threats to validity

Our evaluation was performed over five relatively small Java programs and the results obtained may not transfer to other programs. Since we used a public repository of `C`, `C++`, `C#`, and `Java` programs in which we have no participation, the only bias towards selected programs is our filtering by size and amount of mutations spots (which we already justified due to the cost of performing mutation analysis). Tests suites is another issue. Since we use a rather ambiguous definition of stubborn mutants that needs thoroughly enough test suites we can only do our best to generate those tests suites. We tried to mitigate this issue by using `EvoSuite` to generate a base-line test suite that we then extended according to our best capacity. We used only a subset of available mutation operators for the study; class-level operators where not used because they would only add noise data and are not commonly used; non-basic method-level operators were also opted-out because they are not usually used in experiments using mutants and the complexity of these operators would result in several mutants without any relation to our research questions.

VII. Future work

In the future we would like to improve this operator to take into account boolean expressions found in the code of the method that is currently being mutated and that is reachable from the expression being mutated. A simple example would be the one shown in Figure 11, Where if we apply `BEE` to `retNan` at the last if line we would collect the following expressions to be used (as well as reachable boolean variables, methods and fields):

- `a == b`,
- `a > b`,
- `a < b`.

This improvement uses simple unary boolean expressions and binary relational expressions found in the code with the condition that the expressions inside the unary expression or in the binary relational expressions can neither be another unary expression nor a binary expression. We conjecture that this improvement can allow even further granularity in the weakening and strengthening of boolean expressions.

Another question we have about `BEE` is, even though it cannot directly replace other operators that weaken or strengthen boolean expressions, maybe it can replace the type of faults these operators introduce. If we could replace some operators with `BEE` then we could reduce the amount of mutants required to evaluate a test suite against some types of faults. We could use the work done by René Just et al. in [5] to see what kinds of real faults are related to our new operator and which other operators generate mutants associated with the same real faults.

As we explained previously in Section IV-F, we could change the definition of a stubborn mutant and use the toughness analysis that `muJava++` offers. This would free us from any mistake done while manually checking for stubborn mutants. However, for this to work we would need to rethink the way we generate the test suites. Another way to detect equivalent mutants is using SAT/SMT solvers. In [10] they used symbolic execution and model counting to detect equivalent mutants. However, they point out one problem with this approach which is the complexity of the programs to analyze, if the program is relatively complex their approach will take too much time. If we where to use SAT/SMT solvers in combination with specifications, we would require to write these specifications that in many cases will prevent us to use programs that are to complicate to formally specify, either by their complexity or by limitations in the current tools.

VIII. Conclusions

We have presented BEE, a mutation operator whose main focus is boolean expressions, and that operates by strengthening and weakening such expressions. As we mentioned, although operators that perform this task already exist in the context of mutation testing, e.g. the `COR` operator, these do not provide finer-grained changes in expressions. The rationale behind our new operator is that performing finer-grained mutations in boolean expressions results in more "stubborn" mutants, i.e., more mutants that are difficult to kill.

All the mutation operators used in our experimental evaluation, except for the newly introduced `BEE`, have one common characteristic, they only depend on the available mutation spots. This allows us to anticipate the kind of mutants one will get. `BEE` on the other hand not only depends on the available mutation spots but also on the reachable boolean expressions, variables, fields, and methods. For example, `c = a && b;` will not produce the same mutations than `boolean x = E; ... c = a && b;`, since the variable `x` is only reachable from the second mutation spot. This makes `BEE` less predictable than other operators, in what mutations it could generate. This sensibility to context makes `BEE` more complex and dependent of what kind of context is available for a particular mutation spot. However, this use of context also allows this operator to make changes to boolean expressions that are out-of-reach for other operators. This can play against mutation analysis, as in `ArrayPartition`, where the only reachable method is the postcondition for the method to mutate, generating many equivalent mutants. But can also change some conditions in a way that the final result will only change on very specific conditions as opposed to the simpler alterations of other mutation opertors.

After analyzing the results for the five programs employed in our evaluation, we can now draw some conclusions about our new mutation operator. This operator does not generate a

significant amount of neither stubborn nor equivalent mutants. Although this means that our first research question was negatively answered by this study, the second question can be answered in a more positive way. One main characteristic about BEE is that it depends highly on the available boolean variables, fields and methods to generate mutations. We saw in ArrayPartition that using a method that defines a postcondition of the method that is currently being mutated generates much more equivalent mutants. We think that if we improve upon the expressions used by BEE, (see Section VII), we could manage to improve its impact on mutation analysis. Some positive characteristics of BEE are that it does not generate non-compiling mutants (this has to do with the operators definition), and that the generated mutants that were killed maintained their toughness. An aspect that we would like to examine further is that the mutants generated have an increased amount of mutations spots. This means that after a BEE mutation we could have more places to make further mutations. We would like to see how we could use this aspect, e.g., in program repair contexts, where it may clearly be a benefit. But for using this aspect in mutation analysis we would need to combine mutation operators. Having BEE maintain the difficulty to kill mutants makes it at least as good enough as other operators used, in worst case scenarios, with chances to greatly improve the assessment of test suites in classes with richer sets of boolean variables, fields and methods. We would need to verify if we could use this operator to reduce the set of used ones and keep the same mutation score and toughness values. At present, we can conclude that our operator does not generates non-compiling mutants, the generated mutants are not easy to kill and that by our experiments and the definition of our operator, it generates more subtle changes in boolean expressions that allow a finer-grained strengthening and weakening of conditions.

REFERENCES

[1] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41, April 1978.

[2] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. ACM Transactions on Software Engineering Methodology, 5(2):99-118, April 1996.

[3] Yu-Seung Ma, Jeff Offutt and Yong Rae Kwon. MuJava : An Automated Class Mutation System. Journal of Software Testing, Verification and Reliability, Wiley, 15(2)97-133, June 2005.

[4] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 919-930.

[5] Ren Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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 (FSE 2014). ACM, New York, NY, USA, 654-665.

[6] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering: An International Journal, 10(4):405-435, 2005.

[7] muJava++ github webpage. https://github.com/saiema/MuJava.

[8] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 416-419.

[9] Jeff Offutt and Jie Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. The Journal of Software Testing, Verification, and Reliability, Wiley, 7(3):165-192, September 1997.

[10] Willem Visser, What makes killing a mutant hard? ASE pp. 39-44 2016.

[11] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Softw. Eng. 37, 5 (September 2011), 649-678.

[12] Allen Troy Acree, Jr.. 1980. On Mutation. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA. AAI8107280.

[13] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd. Edition, Cambridge University Press, 2016.

[14] D. Baldwin and F. G. Sayward. Heuristics for Determining Equivalence of Program Mutations. Research Report 276, Yale University, New Haven, Connecticut, 1979.

[15] L. Bottaci and E. S. Mresa. Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability, 9(4):205-232, Dec. 1999.

[16] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In Proceedings of the Summer School on Computer Program Testing, pages 129-148, Sogesta, June 1981.

[17] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. Journal of Systems Software, 38:235-253, 1997.

[18] B. J. M. Grn, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09), pages 192-199, Denver, Colorado, 1-4 April 2009. IEEE Computer Society.

[19] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 1991. Fundamentals of Software Engineering. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[20] Roger S. Pressman. 2001. Software Engineering: A Practitioner's Approach (5th ed.). McGraw-Hill Higher Education.

[21] Pankaj Jalote. 1997. An Integrated Approach to Software Engineering (2nd ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[22] M. Harman, Y. Jia, and B. Langdon. Strong higher order mutation-based test data generation. In 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11), pages 212-222, New York, NY, USA, September 5th - 9th 2011. ACM.

[23] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08), pages 249-258, Beijing, China, 2008. IEEE Computer Society.

[24] R. Just, M. D. Ernst, and G. Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing, volume abs/1303.2784, 2013.

[25] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 701-710, Washington, DC, USA, 2012. IEEE Computer Society.

[26] D. Schuler, V. Dallmeier, and A. Zeller. Efficient Mutation Testing by Checking Invariant Violations. Technique report, Saarland University, Saarbrucken, Telefon, 2009.

[27] D. Schuler and A. Zeller. (Un-)Covering Equivalent Mutants. In Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10), Paris, France, 6 April 2010. IEEE Computer Society.

[28] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. Software Testing, Verification and Reliability, 23(5):353-374, 2013.

[29] Y. Jia and M. Harman. Higher order mutation testing. Information and Software Technology, 51(10):1379 1393, 2009.

[30] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (TSE), 37(5):649678, 2011.

[31] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 433436, 2014.

[32] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 437440, 2014.

[33] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 771-774.