Augmenting Equivalent Mutant Dataset Using Symbolic Execution

Seungjoon Chung
KAIST
Daejeon, Republic of Korea
s.j.chung@kaist.ac.kr

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Mutation testing aims to ensure that a test suite is capable of detecting real faults, by checking whether they can reveal (i.e., kill) small and arbitrary lexical changes made to the program (i.e., mutants). Some of these arbitrary changes may result in a mutant that is syntactically different but is semantically equivalent to the original program under test: such mutants are called equivalent mutants. Since program equivalence is undecidable in general, equivalent mutants pose a serious challenge to mutation testing. Given an unkilled mutant, it is not possible to automatically decide whether the cause is the weakness of test cases or the equivalence of the mutant. Recently machine learning has been adopted to train binary classification models for mutant equivalence. However, training such classification models requires a pool of equivalent mutants, the labelling for which involves a significant amount of human investigation. In this paper, we introduce two techniques that can be used to augment the equivalent mutant benchmarks. First, we propose a symbolic execution-based validation of mutant equivalence, instead of manual classification. Second, we introduce a synthesis technique for equivalent mutants: for a subset of mutation operators, the technique identifies potential mutation locations that are guaranteed to produce equivalent mutants. We compare these two techniques to MutantBench, a manually labelled equivalent mutant benchmark. For the 19 programs studied, MutantBench contains 462 equivalent mutants, whereas our technique is capable of generating 1,725 equivalent mutants automatically, of which 1,349 are new and unique. We further show that the additional equivalent mutants can lead to more accurate equivalent mutant classification models.**

I. Introduction

Mutation testing, proposed by Hamlet [1] and DeMillo et al. [2], is used as a gold standard to assess the quality of a test suite [3]. Given a program under test, developers inject artificial faults to create mutant programs and run a given test suite against these mutants. Such injected faults are intended to mimic common mistakes of developers and a well-made test suite is desired to be able to identify them. According to the two fundamental hypotheses of mutation testing, the *Competent Programmer Hypothesis* and the *Coupling Effect Hypothesis* [2], simple and small syntactic changes are sufficient to cover the fault space. These changes are categorised via mutation operators; typically a single mutation operator is applied to create a mutant. Mutation operators include modifying a binary add operator ($+$) to a subtraction operator ($-$) or replacing a whole conditional statement to $true$ or $false$.

Each mutant is labelled *killed* if at least one test yields a different result from when run against the original program, otherwise the mutant is labelled *live*. The more mutants a given test suite can kill, the higher its fault detection capability is. This is quantified as Mutation Score [4], MS, which is defined as follows:

$$MS = \frac{\#\, of\, killed\, mutants}{\#\, of\, total\, mutants},\ 0 \leq MS \leq 1$$

One widely known challenge in mutation testing is the existence of *equivalent mutants* (hereafter denoted as EM). These are mutants that are syntactically distinct but semantically identical to the original program [5]. Since they are semantically identical to the original program, no test can reveal any difference between EMs and the original program, which misleadingly lowers the mutation score. Consequently, EMs make it difficult to interpret the mutation score: ideally, EMs should be excluded from the computation of mutation scores. The problem is that the program equivalence is undecidable in general, resulting in no highly efficient and automated way of distinguishing EMs from non-equivalent mutants (hereafter denoted as NEMs). Manual inspection is known to be costly, taking about 15 minutes per mutant on average [6] while being prone to error [7]. Many heuristics have been designed based on program invariants [8] or dataflow analysis [9], but they often require non-trivial additional analysis.

Recently, Machine Learning (ML) has been applied to classify EMs from NEMs based on easily collectable features [10]–[12]. While these techniques show promising accuracy, there is the problem of providing a sufficient number of EMs to train the classification models. The cyclic nature of the problem presents a non-trivial challenge: we would like to train a classification model because identifying EMs is difficult; but because identifying EMs is difficult, it is a challenge to build a large enough dataset to train such a model.

Due to the difficulty of identifying EMs, existing datasets are curated with manual inspection, limiting their overall size. As a result, existing ML-based approaches have been trained on these relatively small datasets. Naeem et al. [12] used a dataset with 1,393 EMs and 1,631 NEMs; Brito et al. [11] used 2,005 EMs and 21,406 NEMs, while Peacock et al. [10] used 946 EMs and 284 NEMs. The number of EMs in the most recently introduced EM benchmark, MutantBench [13],

is still of the same order of magnitude: 927 and 462 EMs for C and Java, respectively, out of 4,400 total mutants. While the ML-based techniques for detecting EMs seem encouraging in terms of cost and accuracy, they have been limited by the small dataset size. Thus, we believe it is necessary to repurpose conventional EM detection work and adapt it to EM generation research to boost the performance of ML-based techniques.

This paper aims to propose a way to generate EMs to augment existing benchmarks of EMs. Our approach contains two techniques. First, we adopt the existing symbolic execution-based detection of EMs [14], [15], but for the purpose of augmenting EM datasets. Instead of filtering out EMs for mutation testing, we only choose EMs out of the generated mutants, in a *generate & validate* approach for EMs. Second, more importantly, we use static analysis and symbolic execution to synthesise mutants that are guaranteed to be equivalent, presenting an EM *synthesis* approach. Using both G&V and synthesis approaches, we show that it is possible to almost double the number of EMs in the largest existing benchmark. Our evaluation shows that the augmentation has a positive impact on the performance of ML-based EM classifiers. Further, we highlight some issues (e.g. falsely included EMs) in the manually labelled EMs in current benchmarks, based on the contrast between automatically generated EMs and existing, manually labelled EMs. In summary, the contributions of this paper are as follows:

- We introduce two approaches for augmenting EM datasets used for the training of ML-based equivalent mutant classifiers. Our synthesis approaches are guaranteed to produce equivalent mutants.
- Using the proposed augmentation technique, we present 1,349 new and unique equivalent Java mutants that are not included in the largest existing EM dataset, MutantBench. This addition almost quadruples the number of EMs in MutantBench, which included 462 EMs.
- Our empirical evaluation suggests that EM augmentation can improve the accuracy of ML-based EM classifiers.
- We identify and report some threats to validity in existing, manually labelled EMs.

II. Background

This section presents background information that is relevant to our proposed technique.

A. Symbolic Execution for Mutant Equivalence Check

Symbolic execution is a static analysis technique that represents program inputs as symbols rather than concrete values. Given a max depth, a symbolic execution engine explores every observable program execution path until it reaches the depth scope or the end of the program. Whenever it reaches a conditional branch, it splits the exploration path into two, each corresponding to the case where the branch is evaluated to true and false. If either of the two conditions is infeasible when added up to the path condition collected up to that point, the path gets discarded.

The result of a symbolic execution run can be represented as a set of pairs of path conditions and symbolic program states, with one pair for each feasible execution path. This information can be extracted not only at the end of an execution path (i.e., a leaf state) after the program terminates, but also at any point of a program. In the example below, we only consider the leaf state result. Suppose we run symbolic execution against a program P. At a leaf state, we obtain path condition, PC_i, and the corresponding symbolic return value, r_i. Program semantics can be represented as a first-order logic formula that is the union of pairs of path conditions and corresponding return values, called a *symbolic summary* of a program. If we mutate P into P', we get two corresponding symbolic summaries:

$$P = \bigcup_{i=1}^{n}(PC_i, r_i),\; P' = \bigcup_{i=1}^{m}(PC'_i, r'_i)$$

Note that the number of feasible paths of the two programs (n, m) can differ based on the mutation.

Verifying equivalence of two syntactically different programs can be formulated as proving the equivalence of two symbolic summaries using Satisfiability Modulo Theory (SMT) solvers. We simply check the satisfiability of the following formula:

$$\bigvee_{j=1}^{m}\bigvee_{i=1}^{n}(PC_i \wedge PC'_j) \wedge (r_i \neq r'_j) \tag{1}$$

Intuitively, we consider every combination of path conditions in two given programs, PC_i in P and PC_j in P', and check if the corresponding return values can be different from each other. If the result is SAT, the model obtained from the solver can produce the input that reveals non-equivalence. However, if the result is UNSAT, it means that the two programs are equivalent. Note that, while Equation 1 represents non-equivalence, the opposite is also possible; Baer et al. [15], for instance, encodes equivalence in their formula. Common to our formula and that of Baer et al. is the case of SMT solver not terminating within the given time budget, i.e., timeouts: we interpret this as not being able to know the equivalence explicitly, and discard any such comparisons.

The technique described above is one of the core techniques that consist our work. For ease of understanding, let us provide a concrete example of how symbolic execution can be used to check equivalence between the original program and its mutant.

Listing 1 shows `Min`, a program that returns the minimum of the given two numbers, together with one of its mutants. The mutation has replaced the relational operator, and the mutated line is denoted with the Δ symbol. There are two execution paths in the original program, P_o, as well as the mutant, P_m. The symbolic summaries of two versions of the program are as follows:

$$\begin{aligned} P_o &= (J \leq I, J) \cup (J > I, I) \\ P_m &= (J < I, J) \cup (J \geq I, I) \end{aligned} \tag{2}$$

```java
public int Min(int i, int j)
{
    int min;
    min = i;
    if(j < i){
Δ   if(j <= i){
        min = j;
    }
    return min;
}
```

Listing 1: The `Min` subject program, with ROR mutation applied in line 5.

Using Equation 1, we get the following formula to query the SMT solver:

$$\begin{aligned} &(J \leq I \wedge J < I) \wedge (J \neq J) \\ \vee &(J \leq I \wedge J \geq I) \wedge (J \neq I) \\ \vee &(J > I \wedge J < I) \wedge (I \neq J) \\ \vee &(J > I \wedge J \geq I) \wedge (I \neq I) \end{aligned} \quad (3)$$

In this simple example, it is relatively easy to figure out that all four clauses in Equation 3 are false. The first and the last clause is unsatisfiable since $V \neq V$ is not satisfiable for any integer V; the second clause is equivalent to $(J = I) \wedge (J \neq I)$, which is also unsatisfiable, as the same pattern applies to the third clause.

```
(declare-const I Int)(declare-const J Int)
(assert (or (or (or (and
    (and (< J I)(<= J I))
    (not (= J J)))
  (and
    (and (< J I)(> J I))
    (not (= J I))))
  (and
    (and (>= J I)(<= J I))
    (not (= I J))))
  (and
    (and (>= J I)(> J I))
    (not (= I I))
)))
```

Listing 2: SMTLib2 query.

A corresponding query formulated using the SMTLib2 [16] format is shown in Listing 2. When we query it to the solver, it returns `UNSAT`, verifying that P_m is an equivalent mutant.

B. Machine-Learning Techniques for Classifying EMs

There are a number of existing techniques that propose to build a Machine Learning (ML) binary classifier to distinguish EMs from NEMs. Chekam et al. [17] proposed an ML based approach by using static features, such as mutation location in control flow graphs, to train a binary classification model that can predict likely equivalent mutants. When tested on a collection of C programs, the trained model achieved AUC value of 0.88 and 95%, 35% precision and recall. Naeem et al. [12] extract and utilise 23 features to build such a classifier: features include the location of the mutated node within the program dependence graph, the significance of the mutated node, the entropy of the mutation operator, and features based on the test suite. In an empirical evaluation based on six Java programs, a Gradient Boosted Trees model achieved 88.8% and 90.5% of average precision and recall for prediction of EMs. Brito et al. [11] utilise features extracted from the graph, e.g., whether the mutant is the source/target node of a *primitive arc* [18], which is a directed transition from one basic block to another that is never always followed by other transitions. The trained model achieved an accuracy of 80.30% when classifying 2,005 EMs from 27 C programs. Peacock et al. [10] represent programs as Abstract Syntax Trees (ASTs) and use Recurrent Neural Networks (RNNs) that take ASTs as input to classify EMs. When studied with 582 mutants of five C programs, the proposed ML approach showed a detection accuracy higher than or equal to 90.0%.

Name	Description
AORB	Replace basic binary arithmetic operators with other binary arithmetic operators.
AOIS†	Insert short-cut arithmetic operators.
AOIU	Insert unary arithmetic operators.
AODS	Delete short-cut arithmetic operators.
AODU	Delete basic unary arithmetic operators.
ROR	Replace relational operators with other relational operators, and replace the entire predicate with true and false.
COR	Replace binary conditional operators with other binary conditional operators.
COI	Insert unary conditional operators.
COD	Delete unary conditional operators.
LOI	Insert unary logical operator.
SDL	Deletes each executable statement.
VDL	All occurrences of variable references are deleted from every expression.
CDL	All occurrences of constant references are deleted from every expression.
ODL	Each arithmetic, relational, logical, bitwise, and shift operator is deleted from expressions and assignment operators.
ABS†	Insert absolute operator.
VR†	Replace variable reference with other variable reference.

TABLE I: List of Mutation operators used in our study. Those marked with † are studied for an EM synthesis, whereas the remaining operators are studied for EM Generate & Validate.

III. Equivalent Mutant Augmentation

This section describes the two techniques we propose to augment the EM datasets: 1) Generate & Validate, and 2) EM Synthesis. An overview of our approach that includes the two techniques is depicted in Figure II-B. Table I contains the list of mutation operators we study. Note that we choose only three operators to be studied for EM synthesis because: 1) EMs made of the three operators account for the majority of the EM operator population, 2) explaining EM synthesis techniques for the three appears to be sufficient to convey the main idea, 3) ease of implementation for the three.

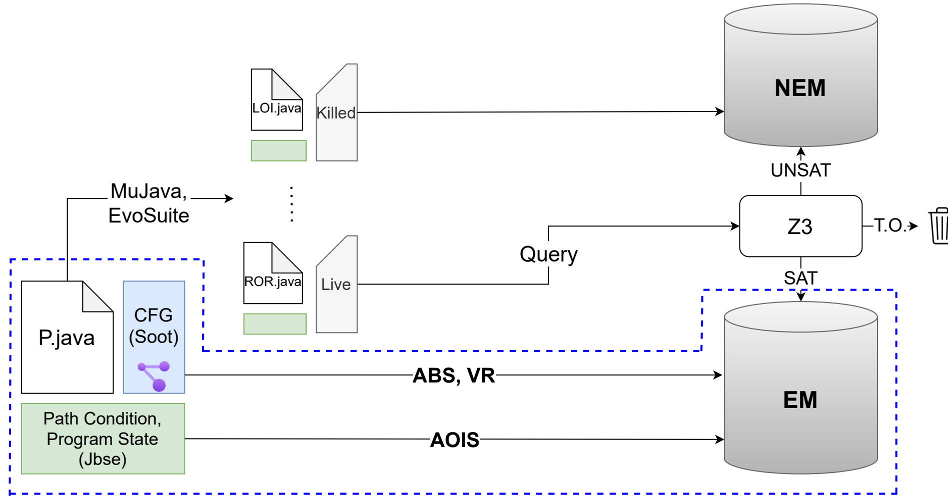

Fig. 1: Overview of our approach. Part enclosed with blue dotted lines represent an EM synthesis, and the rest accounts for G&V.

A. *EM Generate & Validate*

Generate & Validate is essentially reusing existing EM detection techniques to identify EMs: instead of filtering out EMs from actual mutation testing, we intend to augment the existing EM datasets with them. First, we use a mutation testing tool to generate as many mutants as possible. Second, we use test cases to kill and thereby filter out, NEMs: this is to minimise the use of SMT solvers, which can be expensive. After the second step, the remaining mutants are either NEMs that have not been killed due to weak test cases, or EMs. Finally, we apply the symbolic execution-based detection technique, outlined in Section II-A, to identify EMs from the set of remaining mutants.

B. *EM Synthesis*

One downside of the G&V approach is its cost, as we need to process a large number of mutants in the initially generated set. Instead, we propose a novel EM synthesis approach that is driven by the duality between EM detection and generation. Given a program P and a mutant P', detection concerns the question of "if we mutate a *point* in P to make P', would it be equivalent to P?", while generation concerns "to make P and P' equivalent, which *point* in P should be mutated?". Detection focuses on a given mutated point, whereas generation should search for the point to mutate.

While answering the latter question, in general, may still be infeasible (as it can be equally reduced to the question of program equivalence), certain mutation operators allow us to design specific heuristics that would produce EMs. The synthesis of EMs hinges on both the static analysis of the original program and the symbolic summaries of execution paths. The EM synthesis based on symbolic execution is guaranteed pending the capability of the SMT solver. We posit that synthesis can still be more efficient, as we can forego the dynamic validation of many NEMs that are initially generated while sharing the same limitations that are inherent in SMT solvers. Let us below present synthesis heuristics per mutation operator.

1) Synthesis of ABS and VR EMs: Algorithm 1 shows the algorithm for the EM synthesis of ABS (Absolute value insertion) and VR (Variable reference replacement) mutation operators. Using symbolic execution, we obtain the symbolic summary after each statement on every execution path (J_p). From this, we check signs of concrete values (in the case of ABS, lines 10-15) and the existence of another variable with the same concrete value (in the case of VR, lines 25-30). With symbolic values, we similarly decide whether a symbolic value of a variable is positive (in the case of ABS mutation operator, lines 16-21), or identical to another symbolic value (in the case of VR mutation operator, lines 31-36).

These checks allow us to identify places where the application of either ABS (line 37) or VR (line 38) mutation will always result in EMs.

2) Synthesis of AOIS EMs: Here we only consider the AOIS (Short-cut arithmetic operator insertion) mutation operator that inserts shortcut arithmetic operators, but only for the post-increment/decrement. The main idea is to find a variable reference that is thereafter never referenced unless it is re-defined (overwritten) using dataflow analysis.

Algorithm 2 shows the algorithm for the EM synthesis of AOIS mutation operators. We obtain a Use-Definition chain through static analyser (S_p) and combine path information to see if which chain is on a feasible path: if a use of a variable is immediately followed by definitions in all subsequent branching paths, the original use can be mutated with the AIOS mutation (lines 3-16). Note that any infeasible subsequent branches does not affect our synthesis, as the mutation cannot have any impact on that particular execution path (because it cannot be executed). We use the symbolic execution information to determine the path feasibility.

Algorithm 1: ABS and VR Synthesis Algorithm

```
Input: J_p: pairs of line and path information that reaches
       the line of a program p.
Output: S_ABS: pairs of a line and a variable to apply ABS,
        S_VR: pairs of a line and a pair of variables to
        apply VR.
1  foreach line, paths ∈ J_p do
2      positives, equals ← ∅, ∅
3      foreach v_1, v_2 ∈ getLocalVars(p) do
4          isNotPositive[v_1] ← False
5          isNotEqual[(v1, v2)] ← False
6      foreach cond, state ∈ paths do
7          foreach var, val ∈ state do
8              if isNotPositive[var] then
9                  continue
10             else if val is number then
11                 if val ≥ 0 then
12                     positives ← positives ∪ {var}
13                 else
14                     positives ← positives − {var}
15                     isNotPositive[var] ← True
16             else
17                 if checkPositive(val, cond) then
18                     positives ← positives ∪ {var}
19                 else
20                     positives ← positives − {var}
21                     isNotPositive[var] ← True
22         foreach (vr_1, vl_1), (vr_2, vl_2) ∈ state do
23             if isNotEqual[(vr_1, vr_2)] then
24                 continue
25             else if vl_1 is number ∧ vl_2 is number then
26                 if vl_1 = vl_2 then
27                     equals ← equals ∪ {(vr_1, vr_2)}
28                 else
29                     equals ← equals − {(vr_1, vr_2)}
30                     isNotEqual[(vr_1, vr_2)] ← True
31             else
32                 if checkEqual(vl_1, vl_2, pc) then
33                     equals ← equals ∪ {(vr_1, vr_2)}
34                 else
35                     equals ← equals − {(vr_1, vr_2)}
36                     isNotEqual[(vr_1, vr_2)] ← True
37     S_ABS ← S_ABS ∪ bind(positives, line)
38     S_VR ← S_VR ∪ bind(equals, line)
39 return S_ABS, S_VR
```

Algorithm 2: AOIS Synthesis Algorithm

```
Input: linePaths_p: lists of line numbers of which each
       path of a program p consist,
       S_p: a list of basic blocks of p and variable's
       Use-Definition information.
Output: S_AOIS: pairs of a line and a variable to apply
        AOIS.
1  blockPaths_p ← lineToBlock(linePaths_p, S_p)
2  S_AOIS ← ∅
3  foreach var ∈ getLocalVars(p) do
4      isNotCandidate[var] ← False
5      foreach path ∈ blockPaths_p do
6          foreach var_use ∈ getUseInPath(var, path) do
7              line ← getLineNum(var_use)
8              b_use ← getUseBlock(var_use, path)
9              b_def ← getFollowingDefBlock(b_use, path)
10             if isNotCandidate[var_use] then
11                 continue
12             else if existUseBtw(var_use, path, b_use, b_def)
               then
13                 S_AOIS ← S_AOIS − {(line, var_use)}
14                 isNotCandidate[var_use] ← True
15             else
16                 S_AOIS ← S_AOIS ∪ {(line, var_use)}
17 return S_AOIS
```

IV. Experimental Setup

This section describes the experimental setup of our empirical evaluation.

A. Research Questions

We design our experiments to answer the following research questions.

RQ1. Effectiveness *How effective is our approach in its capability of generating unique equivalent mutants?* We use MutantBench as a baseline and analyse the intersection and disjoint sets between EMs in the baseline and those generated by our approach. We check and compare the identities of EMs from both datasets by comparing their ASTs to each other.

RQ2. Utility *Is the created dataset useful for training or evaluating equivalent mutant classification model?* RQ2 concerns the usefulness of our augmentation. Recall that the main purpose of this work is to provide an equivalent dataset generation methodology that can be used for training classifiers. To answer RQ2, we build separate EM classification models, each using a different training dataset: the baseline dataset that is MutantBench, and another dataset that has been augmented by our technique.

RQ3. Replication *Do the generated EMs confirm the existing, manually labelled EMs?* RQ3 qualitatively study the EMs that exist exclusively in either MutantBench or our augmentation set. After manual analysis, we report potential threats in the existing datasets of EMs.

Program	LOC	$\|M\|$	$\|M_T\|$	Test Coverage
ArrayUtils	1582	174	354	94%
Bisect	27	2	3	100%
Bubble	25	1	2	100%
BubbleSort	15	1	3	100%
Day	42	1	15	100%
Defroster	214	2	29	96%
Insert	25	1	4	100%
MathUtils	455	40	113	88%
Mid	27	1	11	100%
Min	10	1	4	100%
Primenum	24	1	2	100%
Profit	40	2	13	100%
QuickSort	50	3	8	100%
Simulator	30	36	10	100%
StringTokenizer	173	13	57	98%
Triangle	41	1	22	100%
Vector3D	135	24	74	100%
WordUtils	239	14	44	99%
XmlFriendlyNameCoder	86	10	13	96%

TABLE II: Subject Java programs from MutantBench: $|M|$ and $|M_T|$ represents the number of program methods and test methods, respectively.

B. Dataset

For comparison to baseline, we use all 19 subject Java programs from MutantBench, whose details are shown in the table II. We apply the following preprocessing. First, we manually investigate the programs and convert void functions to explicitly return the side effects. This process is necessary for the G&V part of our approach requires return value for every function. Second, we remove print statements, as mutating them adds little value to the equivalent mutant dataset.

C. Implementation & Environment

Our EM augmentation approach consists of two techniques: G&V and EM synthesis. For G&V, we use muJava [19] to create as many mutants as possible, using all supported mutation operators. In order to filter out NEMs, we generate test suites by applying EvoSuite [20] to the original version and subsequently running the test cases against the mutants. When the proportion of killed mutants is too low, we manually augmented the test suites. Detail for the prepared dataset and test suite can be found in Table II. We analyse the remaining live mutants using JBSE [21], a symbolic execution tool, and Z3 [22], a backend SMT solver, as described in Section II-A.

For EM synthesis described in Section III-B, we obtain Use-Definition information and Control Flow Graph (CFG) of the given program using Soot [23], a program analysis tool.

For comparison between ASTs of EMs, (described in Section V-A), we use GumTree [24] to represent each mutant to AST and exhaustively check the isomorphism between individual ASTs.

To train classification models, we collect various features related to the generated mutants. Table III contains the features that we use, which are a combination of those introduced by Naeem et al. [12] and by Brito et al. [11]. We build a pipeline to collect relevant features when given a set of an original program and mutants.

All experiments have been performed on a machine with an 8-core Apple M1, equipped with 8GB of unified memory. We use JBSE built with Gradle 7.1.1, which is executed using Java version 1.8. Our classification models are written using scikit-learn [25] version 0.24 and Python version 3.7.

V. Result

A. RQ1.Effectiveness

Table IV shows the number of EMs generated by our approach, in comparison to those in MutantBench. In total, our approach has generated 1,725 EMs. Out of these EMs (denoted by set O in Table IV), there are 1,349 unique EMs that are not present in MutantBench. This effectively increases the total of Java EMs in the benchmark from 462 to 1,811, almost quadrupling the total.

For ArrayUtils, we get rid of some of the generated EMs, as they are simply the same mutations applied to a set of overloaded methods that share the same argument names, but with different type signatures. We regard them essentially as the same EMs and filter them out.

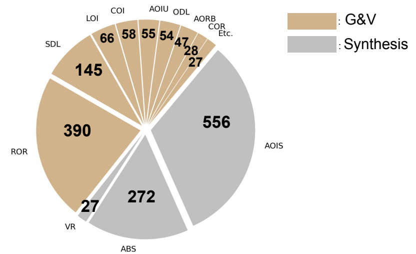

Fig. 2: 855 out of 1,725 EMs are generated through synthesis.

Type	If	Assg.	Block	For	Func. Call	Return	Etc.
Num.	1,012	325	153	16	14	11	194

TABLE V: The number of EMs categorised by the type of mutated statements.

Figure 2 shows the breakdown of the 1,725 generated EMs, focused on the number of EMs created using G&V versus the synthesis approach. Among these, the AOIS operator accounts for the largest portion that is 32%. We also note that 49% of the generated EMs, i.e., 855 EMs, are created by the synthesis approach, which can significantly contribute to the cost reduction of EM benchmark augmentation. We discuss the details of cost reduction in Section VI. Table V shows that our approach can generates EMs in diverse types of statements.

B. RQ2.Utility

RQ2 concerns the comparison of two models: a model trained solely using MutantBench (hereafter denoted as M_{mb}, and a model trained on a dataset that is augmented with our unique EMs (hereafter denoted as M_{aug}). For each trained model, we prepare two test sets: one is collected from both MutantBench and our augmented dataset (hereafter denoted as T_{both}), and the other is collected only from MutantBench (hereafter denoted as T_{mb}). Let us explain the way we form each set.

The training set M_{mb} contains 80% of MutantBench EMs; the training set M_{aug} is M_{mb} augmented with all EMs from O, except for those EMs that we leave out for test sets. Since available EMs are either from MutantBench or our augmented dataset, we pay particular attention to building a fair test set. To construct T_{both}, we randomly sample 31 EMs from each of $M - O$, $M \cap O$, and $O - M$. This results in 93 EMs, which is 20% of the MutantBench EM size, thereby achieving 80:20 split ratio between train and test datasets for M_{mb}.

Note that M_{aug} uses a much larger train set, $O \cup M$: if M_{aug} overfits to EMs in O, having EMs from O in the test set may affect its performance. To mitigate this potential threat, we arrange the second experiment that removes part of the test set brought from O, reducing the size of the test set T_{mb} to 62. Table VI shows the details of datasets for both of the experiments. The result for RQ2 is summarised in Table VII. While the performance of M_{mb} is already high, M_{aug} consistently outperforms M_{mb}. Evaluation on T_{both} shows that M_{aug} outperforms M_{mb}, showing that the

Property	Description	Possible Values
`Operator`	The mutation operator applied to the mutant.	String
`Source_Primitive_Arc`	Indicates whether the mutant is the source node of a primitive arc.	0 or 1
`Target_Primitive_Arc`	Indicates whether the mutant is the target node of a primitive arc.	0 or 1
`Distance_Begin_Min`	The minimum distance from the mutated node to the initial control flow graph node.	Integer
`Distance_Begin_Max`	The maximum distance from the mutated node to the initial control flow graph node.	Integer
`Distance_Begin_Avg`	The average value of Distance_Begin_Min and Distance_Begin_Max.	[0, 1]
`Distance_End_Min`	The minimum distance from the mutated node to the control flow graph end nodes.	Integer
`Distance_End_Max`	The maximum distance from the mutated node to the control flow graph end nodes.	Integer
`Distance_End_Avg`	The average value of Distance_End_Min and Distance_End_Max	[0, 1]
`Complexity`	The number of mutations occurred on the mutated node divided by the total number of mutations.	[0, 1]
`Type_Statement`	The type of the mutated statement.	String (Block, While, For, If, Assignment, Return, Function Call)
`Hub`	Hub value of the mutated node.	[0, 1]
`Authority`	Authority value of the mutated node.	[0, 1]
`PageRank`	PageRank value of the mutated node.	[0, 1]
`Position_Score`	The number of faults revealed by the mutant node divided by the total faults seeded in the mutated node.	[0, 1]
`Operator_Score`	The number of faults revealed by the mutation operator divided by the total seeded faults that belongs to the mutation operator.	[0, 1]

TABLE III: List of collected features. Undermost five features are introduced in [11], and the rest in [12].

Program	$\|O\|$	$\|M\|$	$\|O-M\|$	$\|O \cap M\|$	$\|M-O\|$
ArrayUtils	46	11	**42**	4	7
Bisect	65	32	**33**	32	0
Bubble	23	9	**14**	9	0
BubbleSort	18	19	8	10	**9**
Day	23	14	**9**	14	0
Defroster	747	143	**604**	143	0
Insert	28	19	**11**	17	2
MathUtils	242	20	**228**	14	6
Mid	19	5	**14**	5	0
Min	10	9	**1**	9	0
Primenum	17	4	**13**	4	0
Profit	33	38	7	26	**12**
QuickSort	71	12	**59**	12	0
Simulator	12	11	**12**	0	11
StringTokenizer	89	51	**61**	28	23
Triangle	43	27	**18**	25	2
Vector3D	100	5	**95**	5	0
WordUtils	57	10	**53**	4	6
XmlFriendlyNameCoder	82	23	**67**	15	8
Total	1,725	462	1,349	376	86

TABLE IV: Number of EMs in MutantBench as well as those generated by our approach: O and M indicate sets of EMs from our technique, and MutantBench, respectively.

Dataset	M_{mb}	M_{aug}
$\|Train\|$	400	1,718
$\|T_{both}\|$	93	
$\|T_{mb}\|$	62	

TABLE VI: $|Train|$ denotes the number of EMs in training set for each model. $|T_{both}|$ and $|T_{mb}|$ denote the number of EMs in test set for two experiments to answer RQ2.

Test Set	Model	Accuracy	Precision	Recall	F1
T_{both}	M_{mb}	0.91	0.95	0.87	0.91
	M_{aug}	**0.96**	**0.97**	**0.96**	**0.97**
T_{mb}	M_{mb}	0.94	0.92	**0.93**	0.93
	M_{aug}	**0.95**	**0.97**	**0.93**	**0.95**

TABLE VII: M_{aug} trained using our augmented dataset outperforms M_{mb} trained using MutantBench. Each number is averaged over 20 runs.

augmentation can improve the model performance. Our second experiment using T_{mb} does show that EMs in $O - M$ may bias the model towards those EMs: M_{mb} performs better for T_{mb} than for T_{both}, while M_{aug} performs slightly worse for T_{mb} than for T_{both}. However, the relative differences apart, M_{aug} outperforms M_{mb} in both experiments, showing that augmentation can improve the general performance of EM classification models.

C. RQ3. Replication

We find that there are EMs that belong to the set $M-O$, i.e., mutants that are manually labelled as EMs in MutantBench but are not reproduced by our approach. We sample and manually investigated some of thse to answer RQ3.

Missing Assumptions: Some ABS mutants in MutantBench implicitly assume that the original program receives positive inputs only. An example is a subject program `Profit`. Given an income value, `Profit` calculates the amount of bonus. While normally it is natural to assume the value of income as

being equal to or greater than zero, this is only semantically implied and not specified in `Profit`. As such, our approach does not consider ABS mutations made to these input values as equivalent; instead, it will print out negative input values as a distinguishing input.

Incorrect Equivalence: We report a few EMs in MutantBench to be actually nonequivalent. Here, we briefly mention the cases of `Triangle` and `BubbleSort`. `Triangle` takes three integer sides as input and determine the triangle type. An input `(1, 1, 2)` must be classified as `INVALID` for any EM, but some EMs in MutantBench contradict this. Consider the mutated version of `Triangle`[1] in Listing 3. The mutation in Line 27 causes the program to return `"ISOSCELES"` when input `(1, 1, 2)` is given.

```
1   public String triangle(int a, int b, int c)
2   {
3     int trian;
4     if (a <= 0 || b <= 0 || c <= 0) {
5       return "INVALID";
6     }
7     trian = 0;
8     if (a == b) {
9       trian = trian + 1;
10    }
11    if (a == c) {
12      trian = trian + 2;
13    }
14    if (b == c) {
15      trian = trian + 3;
16    }
17    if (trian == 0) {
18      if (a + b < c || a + c < b || b + c < a) {
19        return "INVALID";
20      } else {
21        return "SCALENE";
22      }
23    }
24    if (trian > 3) {
25      return "EQUILATERAL";
26    }
27    if (trian == 1 && a + b > c) {
  Δ if (trian == 1) {
28      return "ISOSCELES";
29    } else {
30      if (trian == 2 && a + c > b) {
31        return "ISOSCELES";
32      } else {
33        if (trian == 3 && b + c > a) {
34          return "ISOSCELES";
35        }
36      }
37    }
38    return "INVALID";
39  }
```

Listing 3: A mutated `Triangle` marked as equivalent in MutantBench: the mutation in Line 27 will cause the program to accept the input (1, 1, 2) as an isoscleles triangle instead of returning `"INVALID"`

Listing 4 shows a similar mutant[2] for `BubbleSort`. The mutation in Line 8 changes the behaviour of sorting algorithm by duplicating an element in the given array instead of swapping, yet the mutant is marked as an EM.

```
1   public void sort( int[] data )
2   {
3     for (int i = 0; i < data.length - 1; i++) {
4       for (int j = data.length - 1; j > i; --j) {
5         if (data[j] < data[j - 1]) {
6           int temp = data[j];
7           data[j] = data[j - 1];
8           data[j - 1] = temp;
          Δ data[j] = temp;
9         }
10      }
11    }
12  }
```

Listing 4: The `BubbleSort` subject program, with AORB mutation applied in line 8.

[1]This mutant can be found in `dataset.ttl` file of MutantBench by its URI: `f4f5ace09139ab1fc12b096aae6371e9922778b6`.

[2]This mutant can be found in `dataset.ttl` file of MutantBench by its URI: `44994785a8fc0f8bc1a59f6626cde4a81bd0162b`.

Mutation on IO Functions: MutantBench has EMs that mutate `print` statements, which our approach filters out. Thus, such EMs are not present in our results.

Cross-Language Discrepancies Although we only study Java EMs in MutantBench, we have discovered some cross-language discrepancies in some subject programs that exist in multiple languages, which we report here. First is the discrepancy between `Defroster.java` and `Defroster.c`. The Java version has nine class instance variables, all with the default value of 0. However, the C version initialises the corresponding variables with 1. Due to the difference, most of the lines are unreachable in the Java version of the program.

A similar cross-language discrepancy is found between `Prime_num.java` and `Prime_number.c`, both a program that seeks prime numbers less than or equal to a certain number. For the Java version, the termination condition for a for loop is set as `m <= 5`, whereas the corresponding condition for the C version is `m <= 100`. This can potentially lead to different behaviour between the two versions.

D. Threats to Validity

Threats to internal validity concern factors that may have affected the observed outcomes. The proposed approach depends heavily on a number of technical components that can cause such threats, primarily the symbolic execution engine, and the SMT solver. We try to mitigate these threats by choosing JBSE symbolic execution engine and Z3 SMT solver, both widely used tools that have withstood public scrutiny.

Threats to external validity concern the generalisation of our work outside the studied scope. Our claim about the effectiveness of the proposed EM augmentation technique is based on the EM classification model and its features. While only further study can ensure the generalisability of the proposed approach, we tried to provide a fair comparison to existing techniques by adopting model configurations and features used in the previous studies [11], [12].

Another factor that limits the generalisation of our claim is the choice of programs we study. We chose all 19 programs in MutantBench [13]: nine are relatively simple, single function programs, while the rest consist of at least 110 lines of code and five functions. Given the small size of the subject programs, generalization of our approach may be limited by the scalability of its components (i.e., the symbolic execution engine and the SMT solver).

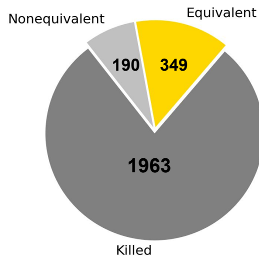

Fig. 3: Among 2,502 AOIS mutants, 349(14%) are equivalent. Our approach reduces the cost wasted on the rest 86%.

VI. Discussion

A. Cost of our approach

One of our technical contributions is the EM synthesis technique: to the best of our knowledge, this is the first attempt at directly creating EMs with no false positives. However, the practicality of our approach depends on its computational cost. Let us analyse the cost-benefit of the synthesis approach.

Schuler and Zeller [6] report that manual identification of an EM can take about 15 minutes on average, which provides the baseline for efficiency (note that manual identification may still produce false positives). We will compare the cost of generating AIOS EMs via G&V and synthesis to this baseline; AOIS is chosen for being the most prevalent mutation.

When applied to 19 studied Java programs, the mutation tool muJava generates a total of 2,502 AOIS mutants. Among these, 539 mutants survived the testing stage; eventually, 349 AOIS mutants are decided to be equivalent. The ratio of each stage's number is shown in a figure 3. Creating 2,502 AOIS mutants, creating and running tests on them, and running symbolic execution followed by one-by-one equivalence comparison takes about 120 minutes for all programs except `Defroster` (additional 485 minutes if we take `Defroster` into account).

In comparison, our synthesis algorithm can directly generate all 349 EMs using static analysis of an original program as well as a single symbolic execution run. It takes less than 10 minutes for 19 programs, even including `Defroster`: the cost reduction is mainly because EM synthesis does not have to query the SMT solver for each mutant.

Suppose we are to construct a pool of AOIS EMs for the 19 subject programs. The cost of manual inspection is estimated as 8,085 minutes (539×15), while our G&V approach takes 485 minutes. The advantage of our synthesis approach is clear, as it takes less than 10 minutes.

B. Limitations and Future Work

In the final stage of G&V, we construct an equivalence query using symbolic execution results that is to be sent to an SMT solver. The query includes an identity clause of return values. While such queries are trivial for primitive types that SMT solvers can take, more complex data types may pose a problem for SMT solvers. Comparing two instances of complex data structures for equivalence may require additional instrumentation that breaks down the data structures into a set of primitive type values. We leave this as future work.

Currently, our technique only aims to generate First Order EMs, as opposed to Higher Order EMs. It is known that non-equivalent First Order Mutants (FOMs) can be combined to create Higher Order Mutants (HOMs) that are equivalent because one FOM masks the other. We expect that categorising such patterns can lead to synthesis techniques for Higher Order EMs. Such techniques will have to consider symbolic relationships between program elements, in a way similar to existing work [26] that exploits dependence relationship to create Strongly Subsuming Higher Order Mutants [27].

VII. Related Work

MutantBench [13] is a open-source EM dataset that can serve as an evaluation benchmark for techniques such as EM classification. We share the same motivation for creating a high-quality dataset of EMs, but aim to provide an accurate and automated EM generation instead of manual labelling.

Several existing work address the challenges EMs present to mutation score using various approaches including static analysis techniques [14], [15], [28]–[30], Higher Order Mutation [31], and compiler optimization [32], [33]. Our approach also relies on symbolic execution, but with the purpose of generating EMs instead of detecting and avoiding EMs. We are motivated by the recent work that aims to build predictive models that can classify EMs based on both static and dynamic features [10]–[12]. By augmenting the existing EM datasets automatically, we hope to improve the existing predictive modelling techniques.

Nonetheless, the approaches we propose are motivated by existing work. Our EM synthesis is inspired by the work of Kintis et al. [9], which identifies problematic data flow patterns that are like to produce EMs when mutated. We actively exploit such patterns to synthesise EMs. Our Generate & Validate technique is a more straightforward application of existing EM detection techniques based on SMT solvers, but we actively report EMs instead of discarding them.

VIII. Conclusion

We introduce an automated way of augmenting existing datasets of Equivalent Mutants (EMs) with the purpose of improving ML-based classification of EMs. Using symbolic execution as well as an SMT solver, our approach can either synthesise a mutant that is guaranteed to be equivalent or actively seek out EMs from generated mutants. The proposed approach is much more efficient than manual labelling of EMs, considering that 49% of EMs we generate are directly synthesised, with no false positives. We evaluate our approach using 19 Java programs in an existing EM benchmark and show that we can quadruple the size of the existing EM dataset with 1,349 unique EMs that are automatically generated. Further, we show that our augmentation can produce EM classification models with higher accuracy.

Acknowledgements

This work has been supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001), and Samsung Electronics (Grant No. IO201210-07969-01).

References

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE transactions on software engineering*, no. 4, pp. 279–290, 1977.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[4] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2008.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275 – 378.

[6] D. Schuler and A. Zeller, "(un-) covering equivalent mutants," in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 45–54.

[7] A. T. Acree, "On mutation," Ph.D. dissertation, Georgia Institute of Technology, USA, 1980, aAI8107280.

[8] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 69–80.

[9] M. Kintis and N. Malevris, "Using data flow patterns for equivalent mutant detection," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 196–205.

[10] S. Peacock, L. Deng, J. Dehlinger, and S. Chakraborty, "Automatic equivalent mutants classification using abstract syntax tree neural networks," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021, pp. 13–18.

[11] C. Brito, V. H. Durelli, R. S. Durelli, S. R. de Souza, A. M. Vincenzi, and M. E. Delamaro, "A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2020, pp. 304–313.

[12] M. R. Naeem, T. Lin, H. Naeem, and H. Liu, "A machine learning approach for classification of equivalent mutants," *Journal of Software: Evolution and Process*, vol. 32, no. 5, p. e2238, 2020.

[13] L. van Hijfte and A. Oprescu, "Mutantbench: an equivalent mutant problem comparison framework," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021, pp. 7–12.

[14] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Employing dynamic symbolic execution for equivalent mutant detection," *IEEE Access*, vol. 7, pp. 163 767–163 777, 2019.

[15] M. Baer, N. Oster, and M. Philippsen, "Mutantdistiller: Using symbolic execution for automatic detection of equivalent mutants and generation of mutant killing tests," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2020, pp. 294–303.

[16] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[17] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.

[18] T. Chusho, "Test data selection and quality estimation based on the concept of essential branches for path testing," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 5, pp. 509–517, 1987.

[19] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 827–830.

[20] G. Fraser and A. Arcuri, "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*, 2011, pp. 416–419.

[21] P. Braione, G. Denaro, and M. Pezzè, "Jbse: A symbolic executor for java programs with complex heap inputs," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1018–1022.

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[24] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[26] S. Oh, S. Lee, and S. Yoo, "Effectively sampling higher order mutants using causal effect," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021, pp. 19–24.

[27] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[28] B. Kushigian, A. Rawat, and R. Just, "Medusa: Mutant equivalence detection using satisfiability analysis," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 77–82.

[29] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson, "Time to clean your test objectives," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 456–467.

[30] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. Le Traon, and J.-Y. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[31] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 701–710.

[32] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2017.

[33] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.