MutRe: Reinforcing Test Suites Using Mutation Testing

20180416 Yeongil Yoon 20180493 Jung In Rhee
20170058 Keonwoo Kim 20170633 Jaegoo Jho
{askme143, ysrheee, keonwoo, jaegoo1199}@kaist.ac.kr

June 22, 2021

https://github.com/CS453-Team-Project/

Abstract

Mutation testing is a useful testing method to estimate the quality of a test suite. However, improving the quality of the test suite requires manual human effort, which is not scalable. To address this issue, we suggest *MutRe*, which creates test cases that kills the survived mutants using symbolic execution. MutRe finds out the input condition that kills the mutant by comparing the symbolic execution results of the original code and the mutated code with the SMT solver. MutRe provides generated test cases in a form of JUnit test codes. There is a path explosion problem from symbolic execution, but we have verified that we can handle code of various patterns.

1. Introduction

Mutation testing is a topic of increasing interest not only in laboratory but also in practice [5]. For example, *Pitest* [4] is a mutation testing tool for real Java development teams.

The developer can consider modifying the test suite according to the result of the mutation test. If the mutation score is low, which means there are many survived mutants, then the developer will want to modify the test suite. However, the mutation score itself is not helpful for the developer to reinforce the test suite. Therefore, mutation testing tools provide reports which contain more detailed information of mutants, but still the human effort is needed to kill survived mutants. Checking each mutants manually does not scale. The more complex and scaled program would have more test cases and mutants to check.

Hence, we propose a test generation and reinforcement tool which utilizes mutation testing: *MutRe*. MutRe generates Junit test code for the Java program. Symbolic execution is used to find inputs that produce different results from the original and mutated programs. Test cases made with these inputs can kill the survived mutants. Because we use the output of the current program as a test oracle, MutRe creates a test suite that captures the current program's behavior well. Therefore, MutRe will also be helpful for regression testing.

2. Backgrounds

2.1. Mutation Testing

Mutation testing is a type of software testing that evaluates test quality by intentionally mutating the code and checking whether the test cases can detect the faults or not. The quality of a test suite is estimated by a mutation score obtained by dividing the number of mutants that found faults by the total number of mutants. If the mute score is low, the developer may consider reinforcing the test suite.

In order for the test suite to kill the mutation, three conditions called RIP conditions must be satisfied: (i) the *reachability condition*, (ii) the *infection condition*, and (iii) the *propagation condition*. The test execution must cover mutated codes. Once the mutated code is executed, it should infect the program state. Also, the infected state should be propagated to an observable result. The developer may find inputs that satisfy RIP conditions and add them into a test suite after mutation testing.

2.2. Symbolic Execution

Symbolic execution is a software testing technique to determine what inputs can cover each part of a program. In symbolic execution, symbolic values are used for inputs instead of actual values. Program variables are replaced with symbolic expressions. By executing with symbolic expressions, each execution path produces a input constraint.

The number of feasible paths grows exponentially as the number of conditional statements increases more in the program. Also, there could be infinite paths, such as an unbounded loop. The cost of a symbolic execution can increase significantly. Moreover, the execution will not terminate for the infinite number of paths or paths with infinite depth. This problem is called path explosion. To cope with this problem, many symbolic execution tools have some constraint setting features such as timeout or heap scope settings.

JBSE [3], the symbolic execution tool for Java we use in this project, provides features that can limit the explore range: *Assertion* and *Assumption*. *Assertions* specify the conditions that must be satisfied for an execution to be correct. JBSE determines whether the symbolic value satisfies all the assertions or not. By applying the assertion, exactly `ass3rt()` function, it is possible to set break line where we want to check whether path condition can reach or not. *Assumptions* specify the conditions that must be satisfied for an execution to be relevant for the analysis. If we use assume function, it is possible to refine the path condition. This concept can be applied for partially blocking path explosion.

2.3. SMT Solver

The *satisfiability modulo theories* (SMT) problem is a decision in first-order logic with equality with additional background theories [2]. Similarly to the *Boolean satisfiability problem* (SAT) solvers, the SMT solvers take a set of clauses to be satisfied and return a satisfying model if the set of clauses is satisfiable, or notifies that the set is unsatisfiable. For instance, we can query to an SMT solver to find a satisfying model of the equations $2x = 6$ and $3y + 2 = 8$ within the integer range $x, y \in \mathbb{Z}$. Then the SMT solver produces a model $x = 3 \wedge y = 2$.

Z3 [6] is one of SMT solvers, developed by Microsoft. It is convenient since it provides APIs to various programming languages, including Python. Moreover, the main theorem prover of JBSE is Z3. Our tool and JBSE both use Z3 to check if a set of path conditions

is satisfiable or not.

3. Methods

To generate test code that kills survived mutants, we used a mutation testing tool *Pitest*, a symbolic execution tool *JBSE*, and an SMT solver *Z3*. Additionally, we used *Javassist* for bytecode level manipulation. MutRe runs mutation testing for the program and extracts metadata and bytecodes from each survived mutant. Then MutRe finds RIP conditions gradually with several times of symbolic executions and SMT solver. SMT solver extracts concrete inputs for the propagation condition if the condition is satisfiable. Finally, MutRe generates test cases that kill mutants.

3.1. Executing JBSE and Parsing Outputs

After executing JBSE once, we get a bunch of *paths*, which are of the following form:

Code 1: The representation of a path in the JBSE output

```
.1.1.1[331]                                          # The name of the path
Leaf state, raised exception: Object[4330]           # The result (leaf state) of the path
Path condition:                                      # The path condition of the path
    {R0} == Object[4326] (fresh) &&
    {R1} == Object[4327] (fresh) &&
    {R2} == Object[4329] (fresh) &&
    ({V3}) >= (0) &&
    (0) >= ({V3})
    where:                                           # The mapping b/w the symbols and the Java variables
    {R0} == {ROOT}:this &&
    {R1} == {ROOT}:s &&
    {R2} == {ROOT}:s.java/lang/String:value &&
    {V3} == {ROOT}:s.java/lang/String:value.length
Static store: { ... }                                # The static store (not in our interests)
Heap: {                                              # The heap
    Object[2824]: {
        Type: (0,[C)
        Length: 16
        Items: {"java.lang.String"}
    }
    # ...
```

```
}
Stack: {                                                # An optional stack dump
    Frame[0]: {
        Method signature: com/cs453/group5/examples/PriQueue:([II)I:solution
        Program counter: 111
        # ...
    }
    # ...
}
```

All information we want to get from the output is the symbol–variable correspondence and path conditions and leaf states written using symbols, which will be used to generate path conditions and leaf states written using variable names only. Therefore, we parsed the output with an emphasis on the types, values, and assigned symbols of variables.

3.2. Finding Reachability/Infection Condition

In this section, the main point is to extract path condition which can reach the target line. For checking whether a path condition reach or not, the assertion concept of JBSE is used. But in this time, it is essential to insert an assertion codes in byte level, so we also use *Javassist* library. More detailed processes are as follows.

3.2.1. Finding Reachability Condition

1. Parsing Mutation Report

   By the result of *Pitest*, it is possible to extract mutation information. Filtering survived mutants and extract information such as mutant id, mutated method and mutated line.

2. Insert Assertion before Mutated Line

   In reachability condition, what we want to do is to extract path condition that can reach the mutated line. We use assertion function for checking reachability. In JBSE, if a path condition fail assertion, then JBSE consider it as violation path.

   Then, our goal is how to insert fail assertion(`ass3rt(false)`) in front of mutated line. we can solve this problem by using *Javassist*. By using *Javassist*, it is possible to insert java codes in a specific line number.

3. Run JBSE and Extract Violation Path

   In this step, we just run JBSE to solve the target byte code that we insert assertion in mutated line. From JBSE results, extract the path conditions which JBSE consider as violation path.

3.2.2. Finding Infection Condition

1. Assume Reachability Condition

   If we find reachability condition, then now we have to refine the path conditions for finding infection conditions. This issue can be solved by inserting assumption (`assume(<R condition>)`) at the top of the mutated method.

2. Insert Assertion after Mutated Line

   This step is very similar with what we did in previous section. But for finding infection conditions, we concentrate on a path condition can reach the next line of mutated line. In other words, if the path condition pass the mutated condition, we can consider it as infection condition. So, in this time we insert `assert(false)` after mutated line.

3. Run JBSE and Extract Violation Path

   This is totally same as previous section. Run JBSE and extract violation path. The extracted path conditions are infection conditions.

3.3. Finding Propagation Condition

This section is about finding the *propagation condition* for a mutant based on *reachability/infection condition* found in section 3.6. By running the JBSE with given *reachability/infection conditions* as assumption, we can get every path that goes through the mutated section. Using the paths found and comparing them with the JBSE result of original code, we can find path conditions that satisfy *propagation condition*. Detailed steps are as follows.

1. Insert assumptions using given conditions

   Using the Javassist library, insert path-condition-assumption using provided *reachability/infection conditions* to the mutant.

2. Run JBSE and parse result

   Run JBSE with path-condition-assumption inserted mutant. Since the given assumption is *reachability/infection conditions*, JBSE will find all the path that satisfies given conditions.

3. Parse JBSE result of the original code

   To find a path condition that satisfies the *propagation condition*, we should compare the leaf state of the mutant paths with leaf state of original code's paths. So parse all the paths that is generated by running JBSE on original code.

4. Compare paths and find propagation condition

   Now we have (1) information of mutant paths that goes through the mutated section and (2) information of all the paths in original code. We compared leaf state of each paths in (1) with every path in (2) to see if the result of two paths are different. If they are different, we use Z3 to solve the intersection of two path's path conditions, which will be the path condition that satisfies the *propagation condition*.

3.4. Solving Kill Conditions

Finally, we found all the path conditions we need. However, there are two more steps to get usable path conditions: translating symbol names into variable names, and translating Z3 functions into Java functions and operators. The path conditions which Z3 gives us are written using Z3 variables and operators, and hence it cannot be injected into the Java code. One may think of a way to used the actual variable names as symbol names. Unfortunately, this method will probably go wrong when there are multiple distinct variables with common names, such as a variable shadowing another. This will obviously ruin the path condition and result in an uncompilable source code.

There is one more reason why such a translation is required. While we comparing path conditions of two different paths, it is possible for a variable to be assigned with symbols with different names in those paths. For instance, suppose the first path has assigned two variables, `a` to `{V1}` and `b` to `{V2}`, while due to some reasons, the second path does not mention `a` at all and hence only `b` is assigned, to `{V1}`. This shows a path-dependency of symbol naming, and hence to compare two path conditions and leaf states, we need to resolve such an issue by translating symbol names into variable names.

Moreover, Java object types are inappropriate to assign corresponding Z3 variables, and hence typical null checking clauses cannot be processed using Z3. Including those,

there are several clauses that do not go well with Z3, so we need to handle them manually. Dealing those conditions with care, we finally have kill conditions in Java syntax.

3.5. Generating Models

After we found the kill conditions, the models are generated to make test suite. The model generation is powered by the SMT solver Z3. We provide path condition clauses, which Z3 can handle, and let it produce a satisfying model if possible. However, the output of Z3, even after translating symbol names and operators, usually not appropriate to be used as a test case. The first reason is that Z3 generates enormously large objects such as an array of length 666780685, or a `java.lang.String` instance of length 1073741824. Unless it is unavoidable to have such a big model, we prefer small and feasible models of which are easier to understand the behaviors. This can be resolved putting those feasibility condition as *soft clauses*, the opposite of *hard clauses*. Hard clauses are one that the model should satisfy, while soft clauses are one that the model does not have to satisfy but it is more preferable to choose a model satisfying soft clauses as many as possible. This kind of problem is called *partial MaxSMT*, as the model is *partially* satisfying the given clauses.

On the other hand, when the kill condition is turned out to be unsatisfiable, then it means that there are no examples (parameters of the method) differentiating two paths, i.e., two paths are equivalent. Therefore, such a pair of paths can be ignored while generating test cases. However, the solver determines the set of clauses to be *undetermined*, which means the solver is too weak to solve the given problem. For example, Z3 cannot provide a solution of $2^x = 4$ where $x \in \mathbb{Z}$, while it seems very easy to find a solution $x = 2$. A workaround to this problem is simply removing those undetermined clauses and find a necessary condition instead. More precisely, when Z3 found a set of clauses undetermined, then we find an example satisfying clauses as many as possible, which becomes a necessary condition to the path condition. Note that in this case, the generated test cases are not guaranteed to differentiate two paths.

3.6. Avoiding Path Explosion

In general, almost all programs use various kinds of loops, arrays and strings. But the JBSE, wich is very fundamental library of our tool, show a serious problem for calculating symbolic path of such programs. Without any refinement, the JBSE calculate the symbolic path as tight as possible. It means the JBSE can calculate symbolic path infinitely

and it will finished when the heap and garbage collection can support those calculations. we call this problem as *Path Explosion*.

To avoid the *Path Explosion*, there is a great concept of *assumption* in JBSE. Simply, to solve this problem we just insert `assume()` at the top of the method. But it is very hard to insert suitable assumption automatically, because it is really hard to select assumption conditions which can be representative of program features. Rather extracting refinement condition automatically, we allow user to make his/her customized assumptions.

Code 2: my_tool_assume.json

```
[
    {
        "class": "com.cs453.group5.examples.Calculator",
        "assumes": [
            {
                "line": 5,
                "assume": "arr.length <= 3"
            }
        ]
    }
]
```

For implementing user customized assumption functionality, we give a json file to users. The structure of json file format is like Code 2. The user can write down the class which they want to execute in `"class"` and make their own assumption. In `"assumes"`, the user have to write down a exact line number and arguments of `assume()`, the conditions user want to refine. For example, if the user execute the MutRe with assumptions above, the program automatically inserts `assume(arr.length <= 3)` in line 5 of target class.

3.7. Converting Data Types

Throughout the program flow, we had to use Z3 many times to solve some satisfiability problems or simplify complicated conditions. The problem is that Z3 has its own data types, we couldn't use the expressions parsed by JBSE directly, which follow the Java syntax. So we had to implement some data conversion method that transforms Java expressions to Z3 expressions and vice versa. This conversion method had led to some limitations of our program, which we will describe in 5.2 **Ones Arisen from Converting**

Data.

Our tool supports all the Java primitives except `short` and `float` (the reason why we do not support these is in 5.2 **Ones Arisen from Converting Data**), primitive arrays and Java classes composed of supported primitives. Z3 has a special data type `BitVector`, which we can use it to represent integer-like n-bit data. MutRe converts all the data types except for `double` to this `BitVector`, since it has to treat primitive types with different size. For floating point numbers `double`, we used `Real` to approximate the floating point arithmetic.

3.8. Generating Human-Friendly JUnit Test Suite

Finally, we have gathered all the information required to generate a full test suite. Nonetheless, there are remaining works to make the output prettier and more legible. It is impossible to deal with a general class instance without knowing the actual content of constructors, getters, setters, and methods dealing with private fields. Therefore, it is quite a reasonable assumption that *there is only one constructor yielding no parameters* and *every member variable is public*, except the following two kinds of entities: array and strings (an instance of the class `java.lang.String`.) The most general thing with these is compactifying any nested structure consisting of arrays, strings, and primitive types. Some examples of such structures are `int[]`, `java.lang.String[]`, and `boolean[][]`.

Code 3: Human-friendly form of the input specification

```
// java.lang.String[] arr;
arr.length = 4;
arr[0].value.length = 3; arr[0].value[2] = '&';    ======>    arr = ["??&", "_", "", ""];
arr[2].value.length = 1; arr[2].value[0] = '_';

// boolean[][] z;
z.length = 3;     z[0].length = 2;                 ======>    z = [[true, false], [], []];
z[0][0] = true;
```

The choices of `'?'`, `[]`, `""`, and `false` are chosen to represent some kinds of emptiness as placeholders, while they can be chosen arbitrarily.

And the inputs generated are converted into JUnit test cases. For each pair of paths, one test case is generated. When the path of the original program returns a value, then

the test case is created using `assertEquals`, and otherwise the test case is created using `assertThrows`.

Code 4: Final JUnit output from MutRe (`NUM_MODELS=4`)

```
@Test
@DisplayName("Original path .1.1.2.1.2.2.2.2.2.2.2.2.2[14] (returned 0) <-> Mutant 2's path
        .1.1.2.2.2.2.1.1.1.1.2.2.2.2.2.2.2[16] (returned 1)")
public void test66() {
    com.cs453.group5.examples.TargetNumber myTargetNumber = com.cs453.group5.examples.TargetNumber();

    int[] numbers;
    int target;

    numbers = [-538052432, 1853521015, 591398826];
    target = 1903393849;
    assertEquals(myTargetNumber.solution(numbers, target), 0);

    numbers = [-538052432, 1853521015, 538052432];
    target = 1903393849;
    assertEquals(myTargetNumber.solution(numbers, target), 0);

    numbers = [-176718471, 1073741824, 176718482];
    target = -1250460295;
    assertEquals(myTargetNumber.solution(numbers, target), 0);

    numbers = [-1250722656, -262361, -838303437];
    target = -1250460295;
    assertEquals(myTargetNumber.solution(numbers, target), 0);
}
```

4. Evaluations

Considering the path explosion problem which is inherent to the symbolic execution technique, we couldn't apply our tool to arbitrary programs. Since the problem was inevitable, we chose to focus on verifying that our tool can deal with various patterns that appear frequently while programming. So we chose the most simple examples of three types of patterns: branch, loop, recursion.

4.1. Branch

In this section, we choose the samples which doesn't have any loop, recursion and complex data type such as array, string. The typical structure of this category is as follows:

Code 5: Calculator.java

```java
public class Calculator {
    public int getSign(int number) {
        int result = 0;
        if (number > 0) {
            result = 1;
        } else if (number < 0) {
            result = -1;
        }

        return result;
    }
}
```

By executing MutRe on `Calculator`, among total **5** survived mutants, we could kill all the **5** mutants.

We found out that for programs with pure branching patterns, MutRe performs well since there is no path explosion problem.

4.2. Loop

The loop is more likely to occur path explosion. In this part, we use user-defined assumption for refinement range of loop. For more details about assumption condition(argument of `assume()`), you want to refer to MutRe repository.

Code 6: Palindrome.java

```java
public class Palindrome {
    public int isPalindrome(String string) {
        int j = string.length() - 1;
```

```
        for (int i = 0; i < string.length() / 2; i++, j--) {
            if (string.charAt(i) != string.charAt(j)) {
                return 0;
            }
        }
        return 1;
    }
}
```

By executing MutRe, among total **9** survived mutants, we could kill **7** mutants.

4.3. Recursion

Lastly, `TargetNumber` class uses BFS algorithm which is based on recursion, and array type. So it is necessary to use proper assumption for executing MutRe.

Code 7: TargetNumber.java

```
public class TargetNumber {
    public int solution(int[] numbers, int target) {
        int answer = 0;
        answer = bfs(numbers, target, numbers[0], 1) + bfs(numbers, target, -numbers[0], 1);
        return answer;
    }
    public int bfs(int[] numbers, int target, int sum, int i) {
        if (i == numbers.length) {
            if (sum == target) {
                return 1;
            } else {
                return 0;
            }
        }
        int result = 0;
        result += bfs(numbers, target, sum + numbers[i], i + 1);
        result += bfs(numbers, target, sum - numbers[i], i + 1);
        return result;
    }
}
```

By executing MutRe, among total **14** survived mutants, we could kill **7** mutants.

5. Discussions

There are some limitations of our tool. They generally fall into one of two categories:

5.1. Ones arisen from Symbolic Execution

1. Difficult to handle loops and lists

   Without proper assumptions on arguments, when the program accesses arbitrary index of a list or when a loop is executed non-constant times. To avoid this, user can use assumption feature of JBSE, but it is also hard to find proper conditions. If the range of refinement increase a little, it can make path explosion anytime. We may avoid this issue using other methods like ones surveyed in [1].

2. Complicated decisions during execution

   The second issue from *Symbolic Execution* is complicated decisions during execution. When there are complicated boundaries, usually associated with loops or recursive calls, JBSE traverses all the cases with some codes by mentioning exact value of each array item or loop conditions.

3. Overall performance issues

   MutRe combines several heavy techniques. During whole process, our tool execute mutation testing, symbolic execution for checking RIP conditions, and cross-checking if there are interaction of path conditions of each resulting pair of paths, from original programmed and a mutant returning, different returned values.

5.2. Ones Arisen from Converting Data

1. Cannot differentiate `short` and `char`

   The first one is Z3 cannot differentiate `short` and `char`, as their common Z3 type, a bit vector of size 16, has no information about signedness. This may be resolved when we carefully revise the implementation, but we choose to not support short.

2. Impossible to compute floating point numbers

The second one is that Z3 seldomly solves when there are clauses containing floating point numbers. Z3 actually support the data type but its solving ability is not that good with floating point numbers. Instead of using floating point numbers, we used exact real numbers. This will change the semantics of the program as follows:

- Floating point error: `assertFalse(0.1d + 0.2d == 0.3d)`
- Precision: `assertTrue(1.0d == 1.0d + 1e-100)`
- Range of representable numbers:
  ```
  double a; if (a > 4.54e2020 && a < 6.7e2021) throw new Exception();
  ```

6. Conclusions

The results above show that our tool performs well for core patterns of programs such as branch, loop, and recursion, when they are properly controlled. Although the path explosion is a quite big problem in **MutRe**, our experiences show that **MutRe** can be a very useful tool if we can control the problem well. Also, by applying user-defined assumption system, user can manually reduce the boundaries of variables that JBSE deal with, by applying reasonable assumption on the program's input.

References

[1] Roberto Baldoni et al. *A Survey of Symbolic Execution Techniques*. 2018. arXiv: 1610.00502 [cs.SE].

[2] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[3] Pietro 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).

[4] Henry Coles et al. "PIT: a practical mutation testing tool for Java (demo)". In: July 2016, pp. 449–452. DOI: 10.1145/2931037.2948707.

[5] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[6] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.