Automated Regression Testing Using Symbolic Execution

Dominykas Barisas, Tomas Milasius, Eduardas Bareisa

Kaunas University of Technology, Software Engineering Department, Studentų St. 50, Kaunas, Lithuania, dominykas.barisas@ktu.lt, tomas1987@gmail.com, edas@soften.ktu.lt

Abstract. The aim of this paper is to describe a way to construct tests which validate that changes made during software evolution did not introduce regression faults. Developers usually run a new version of the program against the same set of tests. In order to achieve this goal, symbolic execution was used for test input generation and full structural code coverage. Moreover, the extension of symbolic execution was developed to increase the quality of tests. As a result, regression faults were detected in the program. The concept of the technique and an example model are presented.

Keywords: regression testing, mutation testing, symbolic execution, automated testing.

1 Introduction

During development and support phases, software is modified to enhance its functionality, detect faults, and adapt it to different platforms. Regression testing is used to identify faults that were introduced when modifying code [1].

A large number of test inputs is generated in order to cover modified parts of the code. Then the tests are executed using generated test inputs on the old and new versions of the code, differences are identified and presented to developer with the details regarding the lines of changed code and the differences [2]. The proposed approach can provide developers with detailed information regarding code coverage and various statistics.

The rest of this paper is organized as follows. Section 2 discusses related work and gives a short overview of existing solutions; section 3 introduces the concept of symbolic execution used in proposed approach. Problem description and possible issues with traditional regression testing approaches are covered in section 0. The following section defines our automated regression testing proposal. Experimental results are provided in section 6. Finally, we conclude and propose future research directions in section 7.

2 Related Work

A lot of research has been done in the area of automatic test case generation, for example an execution of various elements in the program [11] or detection of mutants [12, 19].

Test tools are used for test case execution (for example, Parasoft JTest [13]) and random test input generation. However, random test inputs may not be sufficient to detect different behavior of the new version of program.

Significant amount of research has been done in the area of regression testing in the past few years. Some of approaches [14, 15] rerun test case with the same test inputs and check the outputs of the test case against the captured outputs.

Another approach [16] generates test input set, executes them and collects the return values and object states after the execution of each method under test. The following executions retrieve the same information and check against the initially collected return values and states. Many approaches focus on testing the changed parts of two versions of a software application and takes into account changes related to method return values, object states, and program outputs.

In some cases, finding behavioral differences between two versions of program may not be sufficient and it can be expanded by predicting object state deviations of a changed program [17] or introducing mutation testing [18].

We aim to reach the following goals:

- Detect regressions faults in the program
- Reach as high code coverage as possible
- Improve test input quality by detecting mutants
3 Model Checker

As a model checker for the Java language was chosen Java Pathfinder (JPF) [4, 6] which is built on top of a custom Java Virtual Machine (JVM) and here is used for test input generation. Model checking is done via execution of Java byte-codes, an approach that allows different byte-code interpretations to be developed.

One of the model checking modes in JPF is symbolic execution [5]. Extended interpretation of byte-codes is used to work with symbolic values. Symbolic JPF checks the code for conditional branches incorporating symbolic values, then tries to find out if the branch condition is satisfied for true and false possibilities and identifies values for each branch.

There is a number of helper functions and classes available for JPF, that allow to annotate code, and develop extensions to change and monitor the execution of JPF. One of them is the ability to register Java listeners for various JPF events, for example monitor the execution of a byte-code instruction. Therefore, it allows extensions to access information used internally by JPF. The ability to annotate code and monitoring JPF’s execution is helpful for test generation.

An example of conditioned program and execution tree of the conditioned program [7] is provided in Figure 1.

```java
public int sample(int x, int y, boolean z) throws Exception {
    y -= 2;
    switch(x) {
        case 0:
            if (!z)
                throw new Exception();
            else
                return 0;
        case 1:
            if(x==1)
                return y;
            else
                assert false; // unreachable
        default:
            if(x>y)
                return x;
            else
                return y;
    }
}
```

Figure 1. A snippet of conditioned program and execution tree

4 Problem Statement

In general, mutation describes the modification of a program according to some fault model. Mutation testing is the process of deriving test cases that identify as many mutants as possible. One test input covers one path of the method which may change after the modification of the code and the path will not be executed. Therefore, there will be paths that are never tested and it will cause lower code coverage. Besides, a lot of test inputs and mutants need to be randomly generated in order to cover all paths and catch the mutants. Classic mutation process and our proposed approach are illustrated in Figure 2.
The proposed approach uses symbolic execution which helps to improve code coverage and test input quality by detecting code mutation.

5 The Testing Technique Proposal

The process can be separated into these activities:

- Path condition generation from the source code.
• Test data generation from path conditions. Model checker extension will be developed to improve test data generation [3], which will detect mutation faults as well.
• Execution of generated test cases
• Stored result comparison with the expected results. The test case is considered to have failed in case the result does not match the expected result.

The aim is to produce unit tests because it may be run multiple times and relatively fast. Figure 3 illustrates the described approach with more details.

Proposed concept will address the following faults introduced because of:
• Modification of the application code
• Update of the packets that application is using. The functionality should remain unchanged.
• Changes of the platform

5.1 Mutation Process

Model checking and software testing isn’t the same. By proceeding from model checking to jUnit framework it was found that model checker gives an interval of variable values in order to execute concrete path of the program. However there are cases when the infinite value set is returned and only one value needs to be chosen for the path. Only one choice doesn’t always guarantee that regression faults will be detected. In order to solve this ambiguity, mutation testing will be introduced, which aims to help generating more precise test data [8, 9, 10]. This is explained in section 6.1 with more details and an example.

Main classes involved in test data retrieval and mutated test case generation are presented in Figure 4.

5.2 Limitations

There are several limitations upon this solution introduced by the symbolic execution and constraint solver. First of all, there is no such solution, which fully complies with all the principles of symbolic execution. It is still in development and currently only supports the numeric variable types. Besides, there are many restrictions such as infinite number of states resulting from the ‘for’ loop, recursion and others. However, we are concentrating more on the states returned by symbolic execution rather than addressing issues of symbolic execution.

Secondly, some of the limitations come from the restrictions of the decision engine, called library for constraint satisfaction problems (CSPs) and constraint programming (CP). Interval from -1000000 to 1000000 is valid for variable values, otherwise an error is thrown due to this limitation and is not possible to resolve (for example, restriction $x < -2,000,000$ trigger an error message). These issues are not addressed in our work.

Despite all the above mentioned limitations state mutation was implemented in this work, as well as state joining (program execution path generation), test data generation from the state details, expected result calculation using generated data. Because of these reasons there is no need for manual or random test input generation in order to create a mutant resistant tests. We do not have to rewrite software code and compile it for
each mutant, no need to execute both, the original and the mutated system, for each test case and check whether it catches the mutant. Moreover, we do not have to try and guess test data which protect from mutation and achieve a high code coverage.

6 Test Execution and Test Result Assessment

This section explains how the mutation is performed by giving examples and discussing them in details. Experimental results are provided and compared with different approaches.

6.1 The Need For Mutation

After test data generation we are not sure that it detects changes in the program. Suppose we have this code:

```
import java.math.BigInteger;

public class TestPaths {
   public static void main(String[] args){
      testMe(1, 2, 3);
   }

   public static int testMe(int a, int b, int c) {
      if (a + b > c) {
         return (a + b);
      } else {
         return c;
      }
   }
}
```

After symbolic execution two paths are found and returned:
- \((b_2_{\text{SYMINT}}[0] + a_1_{\text{SYMINT}}[1]) > c_3_{\text{SYMINT}}[0]\)
- \((b_2_{\text{SYMINT}}[0] + a_1_{\text{SYMINT}}[0]) \leq c_3_{\text{SYMINT}}[0]\)

These paths are used to generate corresponding test cases:
- \(\text{testMe}(1,0,0)\) -> Return value: \((a_1_{\text{SYMINT}} + b_2_{\text{SYMINT}})\)
- \(\text{testMe}(0,0,0)\) -> Return value: \(c_3_{\text{SYMINT}}\)

They are entirely correct test cases as all the program paths are executed at least once. However, after the modification of the program these test cases can be no longer adequate as they do not ensure that the faulty change of the program will be found. For example, suppose we had this code: "if \((a + b > c)\)"; and it was changed to "if \((a - b > c)\)" condition. Both the test with \(\text{testMe}(1,0,0)\) and \(\text{testMe}(0,0,0)\) will return a successful test execution value "Passed", although at least one of them should return "Failed" value. Both of these tests will not detect changes in the program and the possible fault.

For these reasons, we introduce mutation testing and trying to predict possible changes in the program. The main idea is that the generated test cases should fit the initial version of the program, but may not be suitable for the mutants (changed versions of the program). In other words, generated test cases will successfully pass using the initial application and fail using mutated application. In order to generate needed test cases, we do not mutate the program itself, but the expressions of execution paths. This approach has the following advantages:

- The process of mutation is simplified because we do not try to replace the original byte-code instructions with mutated instructions. There is no need to modify the software code, compile it and execute a full analysis of the model in order to get the program execution paths and new test cases.
- There is no need to compare execution paths (the initial program and the mutant), so we can combine them and get those test cases that meet the initial version of the program and would not be appropriate to mutants.

Disadvantages of the proposed technique are the following:
- We do not know what the mutant returns. The execution path is mutated, and not the program itself, therefore it may be difficult to determine what values the mutated method will return. However, this is not needed for test case generation and test execution.
- With more complex paths, especially when there are unreachable states in the initial program, it is not possible to have 100% code coverage. One suggestion for the future work could be the extension to detect unreachable code and report it.

6.2 How is the Mutation Carried Out

Once the analysis of the model of system under test (SUT) is finished and the expressions of program execution paths obtained, it can be mutated and connected to the initial expressions, as illustrated in Figure 5.
Figure 5. A process of test data generation

This explained how the test cases are obtained which take the mutants into account and the program changes (possible errors) are detected.

A test case construction algorithm is defined as follows:

MethodsToBeTested : List of methods which should be tested
MethodsInfoList : List of collected information about methods
TestSuite : A set of returned testcases

1. MethodsInfoList ::= []
2. TestSuite ::= []
3. for each method in MethodsToBeTested
4.  MethodInfo = new MethodInfo;
5.  MethodInfo.method = method;
6.  MethodInfo.pathConditions = JPF.findPathConditions(method);
7.  MethodsInfoList.append(MethodInfo);
8. end for
9. for each MethodInfo in MethodsInfoList
10.  for each pathCondition in MethodInfo.pathConditions
11.    mutatedPathConditions = mutate(PathCondition);
12.    mutatedPathConditions = invert(mutatedPathConditions);
13.    for each mPC in mutatedPathConditions
14.      mPC = pathCondition && mPC;
15.      MethodInfo.mutatedPathConditions.append(mPC);
16. end for
17. end for
18. for each MethodInfo in MethodsInfoList
19.  for each pathCondition in MethodInfo.pathConditions
20.    TestCase = generateTestCase(pathCondition.solve());
21.    TestSuite.append(TestCase);
22. end for
23. for each pathCondition in MethodInfo.mutatedPathConditions
24.    TestCase = generateTestCase(pathCondition.solve());
25.    TestSuite.append(TestCase);
26. end for
27. end for
28. return TestSuite;
6.3 Experimental Results

Tests were executed using the following code snippet:

```java
public int testMe(int x, int y, int z, boolean k) throws Exception {
    int res = 0;
    if((15 > y) && (x + 10 < y) && (y > 10) && (y > -x + 5)) {
        switch(z) {
            case 0: res = 0; break;
            case 1: res = x; break;
            default: res = y; break;
        }
    } else {
        if (k) {
            y *= 10;
            if (x > y) {
                res = y + 3;
            }
        } else {
            throw new Exception();
        }
    }
    return res;
}
```

The application was tested three times: first with random test input generation (JTest), second using symbolic execution (JPF) which gives full code coverage and the third with symbolic execution and the extension enabled which takes mutants into account (> , < , <= , >= , == , !== , && , || , ^ , + , - , * , /). There are six conditions and five mutants for each of them, three ampersand and five mathematical operator replacements (6*5+3*2+5*3=51 mutants). The number of detected faults is showed in Figure 6.

![Number of detected faults](image)

**Figure 6. Test result assessment using different test inputs**

The number of detected faults increased from 42 to 51 in the experiment. Test results show that symbolic execution with our extension increases the number of detected possible faults using the same number of test inputs.

7 Conclusions and Future Work

This paper presented a formal technique to the regression testing process satisfying structural code coverage with a higher quality of test data.

Experimental results showed that test data generated with model checker gives a full structural code coverage which increases a number of detected faults in the program comparing to randomly generated test inputs. However, some of mutation faults still remain. This is solved using model checker extension and improved test data generation which increases test case quality and detect more mutants using the same number of test inputs.

Tasks that could be accomplished in the future:
- Combine a number of test cases derived from the different mutants into one test case.
- Create and integrate jUnit extension in test code which keeps track of how many lines of code were executed using the generated tests.
- Add extension that supports complex data structures.
- Add extension that verifies the correctness of code not only according to the returned values, but also based on the inner states of objects or functions.

References


