ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 2011. No. 6(112)
ELEKTRONIKA IR ELEKTROTECHNIKA

T 120 SYSTEM ENGINEERING, COMPUTER TECHNOLOGY
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Automated Regression Testing using Symbolic Execution

D. Barisas, T. Milasius, E. Bareisa
Software Engineering Department, Kaunas University of Technology,
Studentų str. 50, LT−51368 Kaunas, Lithuania, e-mail: dominykas.barisas@ktu.lt

crossref http://dx.doi.org/10.5755/j01.eee.112.6.457

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 [1–18] or to assure that a change, for example a functional enhancement, bug fix, patches or configuration changes, did not introduce new faults. However, a lack of test inputs that exercise a changer behavior is a common issue in regression testing.

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.

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, 21].

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. Other techniques define test input data using OCL constraints [14], which eliminates the need for random data generation.

Significant amount of research has been done in the area of regression testing in the past few years. Some of approaches [15, 16] rerun test case with the same test inputs and check the outputs of the test case against the captured outputs. Tests may be applied for the source code generated from various UML or DSL specifications [17].

Another approach [18] 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 or introducing mutation testing.

In some approaches, symbolic execution is used to improve test input quality. One of such tools for the Java language could be Java Pathfinder [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.

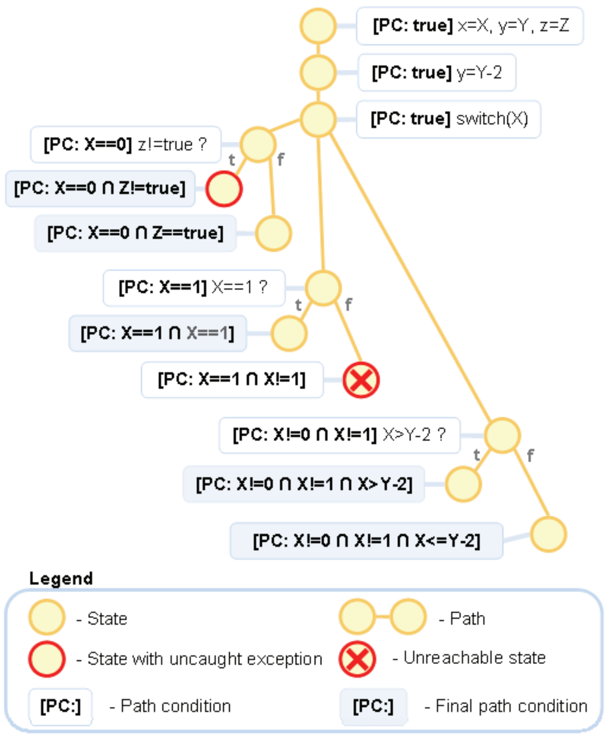


Fig. 1. An example of conditioned program and execution tree

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 [7].

An example of conditioned program and execution tree of the conditioned program is provided in Fig. 1.

We aim to reach the following goals:

1. Detect regression faults in the program;
2. Reach as high code coverage as possible;
3. Improve test input quality by detecting mutants.

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 (Fig. 2) and our proposed approach are illustrated in **Fig.** 3.

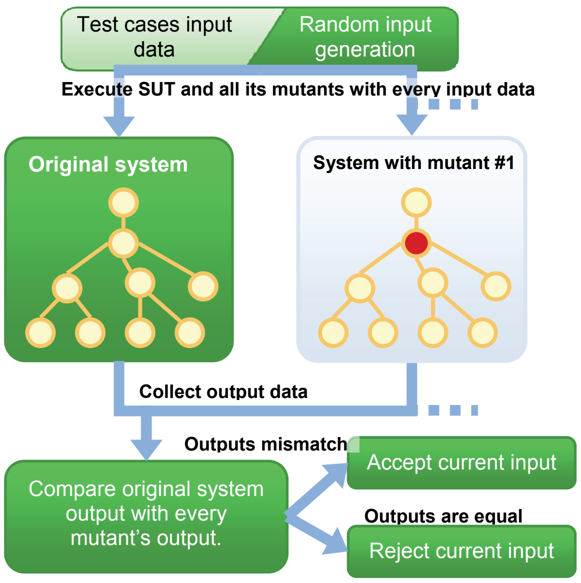
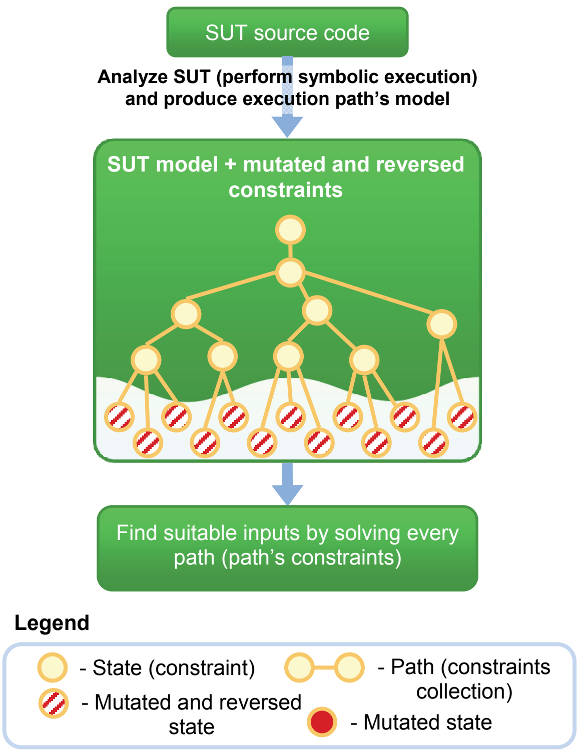


Fig. 2. Classic mutation process

Proposed approach

The proposed approach uses symbolic execution which helps to improve code coverage and test input quality by detecting code mutation.



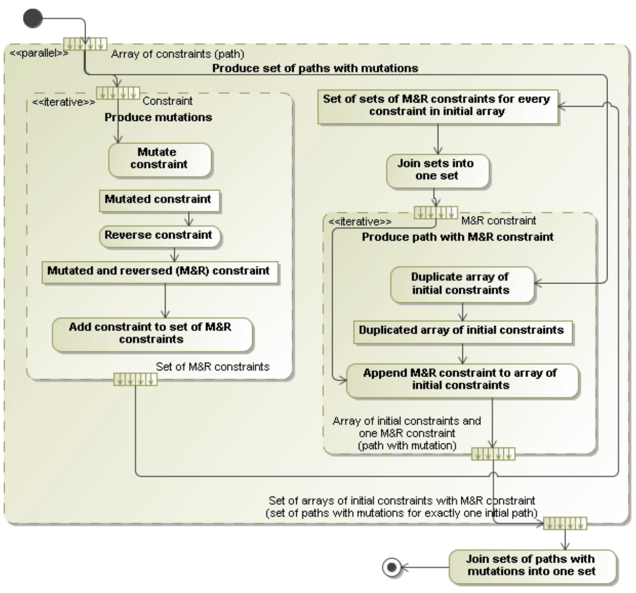
SUT source code
Analyze SUT (perform symbolic execution) and produce execution path's model
SUT model + mutated and reversed constraints
Find suitable inputs by solving every path (path's constraints)
Legend
- State (constraint)
- Path (constraints collection)
- Mutated and reversed state
- Mutated state

Fig. 3. Comparison of classic mutation process and the proposed approach

Testing Technique Proposal and Symbolic Execution

The process can be separated into these activities:

1. Path condition generation from the source code;



<<parallel>>
Array of constraints (path)
Produce set of paths with mutations
<<iterative>>
Constraint
Produce mutations
Mutate constraint
Mutated constraint
Reverse constraint
Mutated and reversed (M&R) constraint
Add constraint to set of M&R constraints
Set of M&R constraints
Set of sets of M&R constraints for every constraint in initial array
Join sets into one set
<<iterative>>
M&R constraint
Produce path with M&R constraint
Duplicate array of initial constraints
Duplicated array of initial constraints
Append M&R constraint to array of initial constraints
Array of initial constraints and one M&R constraint (path with mutation)
Set of arrays of initial constraints with M&R constraint (set of paths with mutations for exactly one initial path)
Join sets of paths with mutations into one set

Fig. 4. A concept of the software testing process

2. Test data generation from path conditions. An extension for symbolic execution will be developed to improve test data generation [3], which will detect mutation faults as well;
3. Execution of generated test cases;
4. 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. Fig. 44 illustrates the described approach with more details.

Proposed concept will address the following faults introduced because of:

1. Modification of the application code;
2. Update of the packets that application is using. The functionality should remain unchanged;
3. Changes of the platform.

Symbolic execution and software testing isn't the same. By proceeding from model checking to jUnit framework it was found that symbolic execution 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 the following section with more details and an example.

Main classes involved in test data retrieval and mutated test case generation are presented in Fig.5.

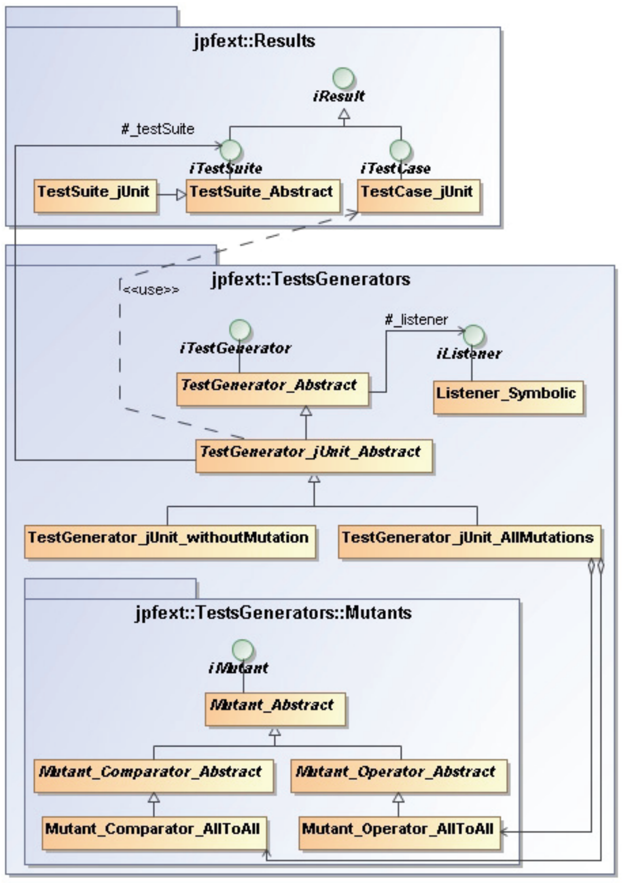


Fig. 5. Symbolic execution extension diagram including mutant generation

Mutation Process and Test Result Assessment

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

```
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:

1. `(b_2_SYMINT [0] + a_1_SYMINT [1]) > c_3_SYMINT [0];`
2. `(b_2_SYMINT [0] + a_1_SYMINT [0]) <= c_3_SYMINT [0].`

These paths are used to generate corresponding test cases:

1. `testMe (1,0,0) -> Return value: (a_1_SYMINT + b_2_SYMINT);`
2. `testMe (0,0,0) -> Return value: c_3_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 `testMe(1,0,0)` and `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:

1. 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;
2. 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:

1. 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;

2. 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.

Once the analysis of the model of SUT is finished and the expressions of program execution paths obtained, it can be mutated and connected to the initial expressions, as illustrated in Fig. 6.

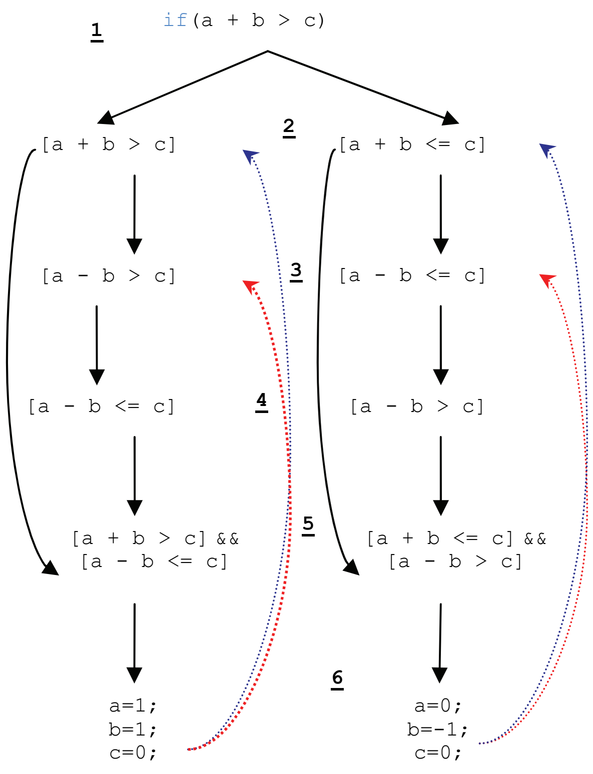


Fig. 6. A process of test data generation

1. Let’s assume we analyze this condition of the application code;
2. We obtain two program execution paths with such conditions;
3. Obtained expressions are mutated. In this case the mutant „+” -> „-“ is applied;
4. Looking for a reverse functions of the mutated, because the new test data should not fit to mutant
5. Reverse functions are connected with the initial functions so that the generated data satisfies both conditions;
6. Concrete test data is found. It satisfies the initial condition (→), but does not satisfy the mutant (→).

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

MethodsInfoList ::= []
TestSuite ::= []
for each method in MethodsToBeTested
      MethodInfo = new MethodInfo;
      MethodInfo.method = method;
      MethodInfo.pathConditions =
      JPF.findPathConditions(method);
      MethodsInfoList.append(MethodInfo);
end for
for each MethodInfo in MethodsInfoList
      for each pathCondition in
      MethodInfo.pathConditions
                 mutatedPathConditions =
                 mutate(PathCondition);
                 mutatedPathConditions =
                 invert(mutatedPathConditions);
                 for each mPC in mutatedPathConditions
                            mPC = pathCondition && mPC;

      MethodInfo.mutatedPathConditions.append(mPC);
                 end for
      end for
end for
TestSuite = generateTestCases(
      MethodInfo.pathConditions,
      MethodInfo.mutatedPathConditions);
return TestSuite;
```

For the experimental research tests were executed using the following code snippet:

```
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 Fig. 7.

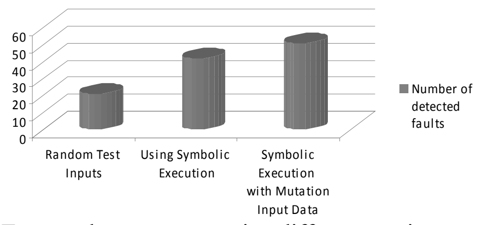


Fig. 7. Test result assessment using different test inputs

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.

Conclusions

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 symbolic execution 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 symbolic execution 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:

1. Combine a number of test cases derived from the different mutants into one test case;
2. Create and integrate jUnit extension in test code which keeps track of how many lines of code were executed using the generated tests;
3. Add extension that supports complex data structures;
4. 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

1. **Orso A., Xie T.** BERT: BEhavioral Regression Testing // Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008). – Seattle, Washington, 2008. – P. 36–42.
2. **Oezbek C.** Introducing Automated Regression Testing in Open Source Projects // Proceedings of the OSS'2010. – Notre Dame, IL, 2010. – P. 361–366.
3. **Visser W., Pasareanu C., Khurshid S.** Test Input Generation with Java PathFinder // Proceedings of ISSTA'2004. – Boston, MA, 2004. – P. 97–107.
4. **Khurshid S., Pasareanu C. S., Visser W.** Generalized Symbolic Execution for Model Checking and Testing // Proceedings of TACAS'2003. – Warsaw, Poland, 2003. – P. 553–568.
5. **Pasareanu C. S., Visser W.** Symbolic Execution and Model Checking for Testing // Verification Conference, 2007. – P. 17–18.
6. **Artho C., Drusinsky D., Goldberg A., Havelund K., Lowry M., Pasareanu C., Rosu G., Visser W.** Experiments with Test Case Generation and Runtime Analysis // Proceedings of the abstract state machines 10th international conference on Advances in theory and practice. – Taormina, Italy, 2003. – P. 87–108.
7. **Walkinshaw N., Bogdanov K., Ali S., Holcombe M.** Automated discovery of state transitions and their functions in source code // Software Testing, Verification and Reliability. – Wiley InterScience, 2007. – P. 99–121.
8. **Bybro M, Arnborg S.** A Mutation Testing Tool for Java Programs. Thesis, Department of Numerical Analysis and Computer Science, Nada at the Royal Institute of Technology. – KTH, Sweden, 2003. – P. 6–22.
9. **Offutt J., Untch R. H.** Mutation 2000: Uniting the orthogonal // In Mutation 2000 Conference: Mutation Testing in the Twentieth and the Twenty First Centuries. – San Jose, CA, 2000. – P. 45–55.
10. **Kim S., Clark J. A., McDermid J. A.** Class Mutation: Mutation Testing for Object–oriented Programs // Proceedings of the Net.ObjectDays Conference on Object–Oriented Software Systems, 2000. – P. 9–12.
11. **Gupta N., Mathur, A.P., Soffa, M.L.** Generating Test Data for Branch Coverage // ASE'00, 2000. – P. 219–227.
12. **DeMillo R., Offutt, J.** Experimental results from an automatic test case generator // ACM TOSEM, 1993. – P. 109–127.
13. **Taneja K., Xie T.** DiffGen: Automated Regression Unit–Test Generation // Automated Software Engineering (ASE'2008), 2008. – P. 407–410.
14. **Packevičius Š., Ušaniov A., Bareiša E.** The Use of Model Constraints as Imprecise Software Test Oracles // Information Technology And Control. – Kaunas, Technologija, 2007. – Vol. 36. – No. 2. – P. 246 – 252.
15. **Saff D., Artzi S., Perkins J. H., Ernst M. D.** Automatic test factoring for Java // In Proc. IEEE International Conference on Automated Software Engineering (ASE'2005), 2005. – P. 114–123.
16. **Orso A., Kennedy B.** Selective capture and replay of program executions // In Proc. International ICSE Workshop on Dynamic Analysis (WODA'2005), 2005. – P. 29–35.
17. **Pavalkis S., Nemuraitė L., Tarvydas P., Noreika A.** Specification of Finite Element Model of Electronic Device using Model–driven Wizard–based Guidance // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 2(98). – P. 59–62.
18. **Xie T.** Augmenting automatically generated unit–test suites with regression oracle checking // In Proc. European Conference on Object–Oriented Programming (ECOOP'2006), 2006. – P. 380–403.

Received 2011 04 12

D. Barisas, T. Milasius, E. Bareisa. Automated Regression Testing using Symbolic Execution // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 6(112). – P. 101–105.

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. Ill. 7, bibl. 18 (in English; abstracts in English and Lithuanian).

D. Barisas, T. Milašius, E. Bareiša. Programinės įrangos regresinis testavimas naudojant simbolinius vyksmus // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2011. – Nr. 6(112). – P. 101–105.

Pateikiamas būdas, kaip aprašyti kūrimą testų, kurie tikrina, ar dėl programinės įrangos pakeitimų neatsirado regresinių klaidų. Programuotojai paprastai vykdo naują programos versiją naudodami tą patį testų rinkinį. Siekiant šio tikslo modelio tikrinimas buvo naudojamas testiniams įėjimams generuoti, šitaip siekiant padengti visas programos būsenas. Modeliui tikrintii buvo sukurtas praplėtimas, kuris padėjo pagerinti testų kokybę ir surasti daugiau regresinių klaidų. Perteikta pagrindinė idėja ir pavyzdinis modelis. Il. 7, bibl. 18 (anglų kalba; santraukos anglų ir lietuvių k.).