AUTOCODECOVERGEN: PROTOTYPE OF DATA DRIVEN UNIT TEST GENRATION TOOL THAT GUARANTEES 100% CODE COVERAGE FOR C#

Arpit Christi[1]

[1]Department of Information Science, New Horizon College of Engineering, Bangalore, India
arpitchristi@ieee.org

ABSTRACT

Unit testing has become an integral part of software development process as it ensures all your individual functions behave in a way they should behave. Obtaining 100% code coverage out of unit test suit is essential to ensure that all code paths are covered. This paper proposes a tool AutoCodeCoverGen that guarantees 100% code coverage. It also discusses the core idea behind tool and evolution of the tool. Also, it shows how the tool differs from existing tools and what benefits it reaps for its users.

KEYWORDS

Unit testing, code coverage, Automated software testing

1. UNIT TESTING AND CODE COVERAGE

Unit Testing your code or methods has become an essential development activity, almost all well-defined software processes show a provision for unit testing [1][2]. Unit tests test a method or a block of code or a unit as opposed to scenario tests that test a user action or a defined scenario although development teams are using unit tests to automate their scenario testing [3]. For our further discussion we are going to stick to the pure definition of unit test, the single unit tester. Unit testing has become so obvious and essential that there exists number of frameworks to aid unit test writing, maintaining and running. MSTest, JUnit and NUnit are good examples of such frameworks/tools that are well recognized among developer community and have added good value to software development process [4][5][6][7]. When development teams go through the extra pain of writing, maintaining and running unit tests, they want to know what percentage of the total code is covered by their efforts, giving birth to the idea of code coverage [8][9]. Most of the professional tools have built in support to measure code coverage [10][11]. Assuring 100% code coverage is ultimate goal of any unit test suite, so that developers can be sure of all paths of their codes being covered by unit testing effort.

2. AUTOMATICALLY GENERATING UNIT TESTS FROM CODE

For our further discussion we will take Microsoft® unit test as our example unit test framework, but similar ideas can be applied to other unit test frameworks. Most of the professional tools can look at the code and generate unit test stubs for your code and then filling details and asserting for correctness is left on developers. We will use following code example for our further discussion. Code is very self-explanatory and we are not concerned about functionality but we are interested in demonstrating unit testing against this code.

DOI : 10.5121/acij.2012.3508

```
public class MathFunctions
   {
      public int a;
      public int b;
public int Greater(int a1, int a2)
      {
return a1 >= a2 ? a1 : a2;
      }
      public int Greater(int a1, int a2, int a3)
      {
         if (a1 >= a2)
         {
            if (a1 >= a3)
            {    return a1;         }
            else
            {    return a3;         }
         }
         else
         {
            if (a2 >= a3)
            {    return a2;          }
            else
            {    return a3;          }
         }
      }
      public intGreaterBetweenAandB()
      {
return a >= b ? a : b;
      }
   }
```

Any tool used to generate unit test against this code will yield 3 unit test stubs, one for each method and to provide testing and asserting within these stubs will be left on developer. So, when automatically generating our test methods for test suite, we are left with 3 stubs for these methods. Let's assume that name of these stubs methods are Greater1Test, Greater2Test and GreaterBetweenAandBTest for Two Greater methods and one GreaterBetweenAandB method respectively. These three unit test stubs will be loaded with some initial values for testing that user can override. One such automatically generated unit test class with stub methods is shown below, generated using visual studio 2010 and MSTest [5].

```
 [TestClass()]
   public class MathFunctionsTest
   {

      [TestMethod()]
      public void GreaterTest()
      {
MathFunctions target = new MathFunctions();
int a1 = 5;
int a2 = 3;
int expected = 5;
```

```
int actual;
            actual = target.Greater(a1, a2);
Assert.AreEqual(expected, actual);
        }

        [TestMethod()]
        public void GreaterTest1()
        {
MathFunctions target = new MathFunctions();
int a1 = 5;
int a2 = 4;
int a3 = 3;
int expected = 5;
int actual;
            actual = target.Greater(a1, a2, a3);
Assert.AreEqual(expected, actual);
        }


        [TestMethod()]
        public void GreaterBetweenAandBTest()
        {
MathFunctions target = new MathFunctions();
target.a = 5;
target.b = 3;
int expected = 5;
int actual;
            actual = target.GreaterBetweenAandB();
Assert.AreEqual(expected, actual);
        }
    }
```

As one can see, we supply only one set of input for each of the test. We can create data driven tests and generate multiple set of input/output to execute unit test with more inputs [12]. To ensure 100% code coverage for our current test suite, we need 2 set of inputs for GreaterTest and GreaterBetweenAandBTestand 4 inputs forGreaterTest1 to ensure all code paths are covered by the unit testing effort (considering "if-else" paths). One can copy paste these tests, rename them and manually provide all required set of inputs to corresponding tests, but that will be practically impossible as real test suite will contain 1000s of tests and code may possibly take millions of paths.

3. EXISTING TOOLS

There exist tools that generate unit tests so that 100% code coverage is guaranteed. PEX is one such tool developed by Microsoft research [13]. We found PEX to be very thorough and useful tool and we highly recommend PEX to anyone who wants to automatically generated unit tests for their test suite to ensure 100% code coverage. PEX first generates parameterized unit test (also known as PEX methods) for each method in the code and then generate unit tests such that all code paths are covered. PEX unit test generation is applied to MathFunctions class and below is the generated PEX methods and unit tests [14].

```
 [PexClass(typeof(global::MathFunctions.MathFunctions))]
  [TestClass]
  public partial class MathFunctionsTest
  {
      [PexMethod]
      public int Greater(
          [PexAssumeUnderTest]global::MathFunctions.MathFunctions target,
int a1, int a2
      )
      {
int result = target.Greater(a1, a2);
          return result;
      }

      [PexMethod]
      public int Greater01(
          [PexAssumeUnderTest]global::MathFunctions.MathFunctions target,
int a1,
int a2,
int a3
      )
      {
int result = target.Greater(a1, a2, a3);
          return result;
}

      [PexMethod]
      public                                                                    int
GreaterBetweenAandB([PexAssumeUnderTest]global::MathFunctions.MathFunctions
target)
      {
int result = target.GreaterBetweenAandB();
          return result;
       }
  }

public partial class MathFunctionsTest
  {
      [TestMethod]
      [PexGeneratedBy(typeof(MathFunctionsTest))]
      public void Greater609()
      {
inti;
          global::MathFunctions.MathFunctions s0
            = new global::MathFunctions.MathFunctions();
i = this.Greater(s0, 5, 3);
Assert.AreEqual<int>(5, i);

      }

      [TestMethod]
      [PexGeneratedBy(typeof(MathFunctionsTest))]
      public void Greater938()
```

```
      {
inti;
          global::MathFunctions.MathFunctions s0
            = new global::MathFunctions.MathFunctions();
i = this.Greater(s0, 3, 5);
Assert.AreEqual<int>(5, i);

      }
   }
```

As one can see, PEX generated 3 pex methods (Greater, Greater01 and GreaterBetweenAandB) and then created 8 unit test methods for the code. For simplicity, we have just shown unit tests for "Greater" pex method, Greater609 (for a > b) and Greater938 (for a < b). PEX and many other tools use symbolic execution or some variance of symbolic execution where an execution tree is created to represent the possible paths for code and then generate input in such a way that all these possible execution paths are satisfied [15][16][17].

4. PROPOSED TOOL AND ITS FEATURES

We are proposing a tool "AutoCodeCoverGen" (Abbreviation of Automatic Code Coverage Generator) that is very similar to PEX but achieves certain different goals that make natural sense to some test writing teams. First we are going to propose the tool and its features and in next section we will discuss pros and cons of our tool with respect to PEX. It's worthwhile to mention that AutoCodeCoverGen is not a fully implemented tool but it's at a prototype level. AutoCodeCoverGen also uses symbolic execution, a well trusted and well tested method to generate automatic unit tests. It generates execution tree with possible paths and conditions of entry to those paths. Then it uses a random data generator in an intelligent way to quickly find out a sequence of inputs that satisfies one tree branch and it will repeat it for all tree branches until all tree branches are exhausted. Time complexity associated with the data generation for AutoCodeCoverGen is order of n where n is number of nodes in execution tree. A single branch starting from root node to one of the leaf nodes represent one set of input and total number of set of inputs required for 100% code coverage are total branches from root to leaf nodes. Once the tool knows all set of inputs it creates a data source with single table where each row represents a single set of inputs. Currently tool emits an xml file as data source with Table1 being the table. Let's see how tool works with Greater(a0,a1,a2) function.

As shown in figure 1, it creates the execution tree and generates set of conditions so that execution tree can be traversed and exhausted. Then it creates an xml file, Greater.xml with single table Table1 and insert rows into these table. Content of Greater.xml file is shown below.

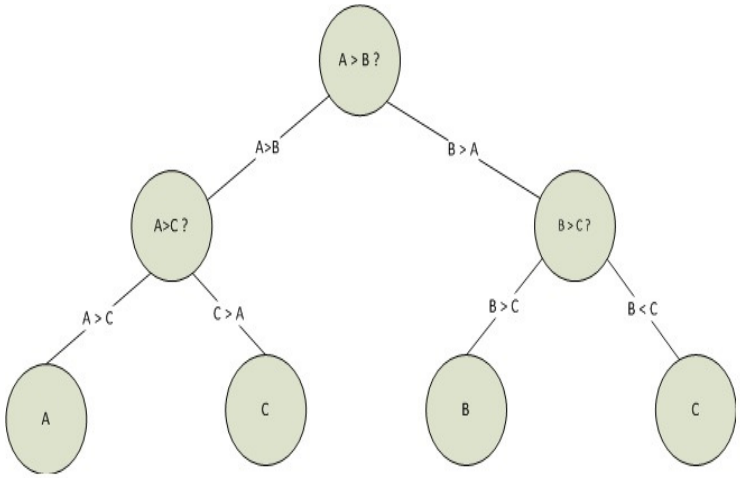

Figure 1. Symbolic execution tree for Greater(a,b,c)

```
<?xml version="1.0" encoding="utf-8"?>
<Table1>
<Row>
<a1>10</a1><a2>5</a2><a3>4</a3><ans>10</ans>
</Row>
<Row>
<a1>10</a1><a2>5</a2><a3>13</a3><ans>13</ans>
</Row>
<Row>
<a1>4</a1><a2>7</a2><a3>3</a3><ans>7</ans>
</Row>
<Row>
<a1>4</a1><a2>7</a2><a3>9</a3><ans>9</ans>
</Row>
</Table1>
```

After generating the xml file and adding it to required project/folder, it creates automated unit test, single unit test for each of the method. So, outcome is very similar to test methods in section 2 rather than pex methods mentioned in section 3. The difference here is, instead of directly assigning values to input parameters, it assigns values from DataRow. Almost all unit test framework has some ways to do data driven testing, inject data from rows of a table [6][7][12]. Microsoft unit test also provides data driven testing and tool uses this ability to generate unit tests [12]. It also associates generated xml to the data source of generated test. The resulting unit test method for Greater(a0,a1,a2) using AutoCodeCoverGen is shown below.

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
"|DataDirectory|\\Greater.xml", "Row", DataAccessMethod.Sequential),
DeploymentItem("MathFunctions.Tests01\\Greater.xml"), TestMethod()]
        public void GreaterTest1()
```

```
      {
MathFunctions target = new MathFunctions();
int a1 = int.Parse(TestContext.DataRow["a1"].ToString());
int a2 = int.Parse(TestContext.DataRow["a2"].ToString());
int a3 = int.Parse(TestContext.DataRow["a3"].ToString());
int expected = int.Parse(TestContext.DataRow["ans"].ToString());
int actual;
          actual = target.Greater(a1, a2, a3);
Assert.AreEqual(expected, actual);
      }
```

Now when this test is executed either individually or as part of test suite, it runs 4 times, running once for each row in Greater.xml file, ensuring 100% code coverage [18]. Let's look an example where instead of using direct parameters, method uses state of an object in its decision making. GreaterBetweenAandB is one such function that determines greater between two class variables and hence uses state of the object of class MathFunctions. The xml file, GreateBetweenAandB.xml and generated test method by AutoCodeCoverGen is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<Table>
          <Row><a>4</a><b>3</b><ans>4</ans></Row>
          <Row><a>67</a><b>98</b><ans>98</ans></Row>
</Table>

[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
"|DataDirectory|\\GreaterBetweenAandB.xml",     "Row",     DataAccessMethod.Sequential),
DeploymentItem("MathFunctions.Tests01\\GreaterBetweenAandB.xml"), TestMethod()]
      public void GreaterBetweenAandBTest()
      {
MathFunctions target = new MathFunctions();
target.a = int.Parse(TestContext.DataRow["a"].ToString());
target.b = int.Parse(TestContext.DataRow["b"].ToString());
int expected = int.Parse(TestContext.DataRow["ans"].ToString());
int actual;
          actual = target.GreaterBetweenAandB();
Assert.AreEqual(expected, actual);
      }
```

If there is one xml generated for each test method, number of xml files will be very large so tool creates single xml file with different table names and uses correct table for corresponding test, if it is applied on entire module or project as a whole but if it is applied for an individual method it creates one xml file for that method separately. In a practical scenario, test suite owner or developer will use the tool against entire project and hence they have one to one correspondence between number of classes and number of test data xml files.

5. ADVANTAGES AND DISADVANTAGES OF PROPOSED TOOL

Though PEX is very thorough and complete tool, one of the issues we have using it is how tests are created, the correlation between number of test methods and number of corresponding

methods. It depends on the symbolic execution tree and number of decision paths available in your code. With real life projects, number of decision paths available can be huge in one method and hence number of tests created can be massive. It's very difficult to understand, explain and maintain a test suite where changing/adding few methods in code can add large number of tests in test suite. Though developers may be fully aware of internal behaviour of PEX or other tool, from a managerial view point or from a maintenance view point, it doesn't make natural sense to have test suite that grows out of control. AutoCodeCoverGen strictly maintains one test method per code method, that makes it easy for a non-developer person to understand it and it also becomes easy to maintain it. With PEX and other tools, test data is embedded within test itself while AutoCodeCoverGen achieves absolute separation between test data and test method by keeping test data into separate xml file or other data source and then calling data from test method whenever necessary [19]. It makes it very easy to add more test data and have your tests run against more test cases and it doesn't require adding/updating your tests and recompiling them. User can just keep on adding test cases by adding new rows to xml file associated with test and whenever test runs, it runs for all the rows in xml file, picking up new test cases added. AutoCodeCoverGen achieves separation of test data and test method by adding one xml file per class and adding test data to this xml file. Just like other tools have the problem of large number of tests with large number of decision making branches and variables that participate in this decision making, AutoCodeCoverGen may end up creating and adding massive xml files in project that may be both massive in size and numbers. Also, with AutoCodeCoverGen, tests may take longer to run, as with other tools, test data and test lives together, they both will be loaded in the memory together and hence test data is accessed by test very fast while AutoCodeCoverGen requires extra time consuming step of reading and accessing data from xml file, making test data access slower and making it longer for tests to run.

6. FUTURE WORK AND CONCLUSION

As we mentioned previously in the discussion, AutoCodeCoverGen is not a completely implemented tool but it exists as a small prototype command line tool that currently handles only decision making code(if, if-else, if-elseif-else, while loop etc.). We want to come up with more thorough command line tool and make it available for developer community for free usage. Apart from handling decision making code, most of the commercial tool provides other useful constructs like exception handling verification, verification against pre-conditions and post-conditions etc. by using methodologies other than symbolic execution. We would like to add these capabilities to AutoCodeCoverGen at some point of time. This paper discusses how activity of unit testing fits into overall software development process and significant of code coverage on unit test suite. It discusses the process and tools for automatically generating unit tests. It then discusses one of the tools, PEX, for automatically generating unit tests. Finally it culminates itself into proposing a new tool, AutoCodeCoverGen, to generate unit tests that guarantee 100% code coverage. Relative advantages/disadvantages of AutocodeCoverGenwith respect to other tools are also discussed. Overall, AutoCodeCoverGen proposes a philosophy of keeping test data separate from test and keeping one to one correspondence between test method and actual method while automatically generating unit tests to guarantee 100% code coverage.

REFERENCES

[1] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[2] Williams Laurie, KudrajavetsGunnar &Nagappan Nachiappan (2009)"On the effectiveness of unit test automation at Microsoft", Microsoft® Research.

[3] KanerCem (2003)"The power of 'What If…' and nine ways to fuel your imagination: CemKaner on scenario testing", Software Testing and Quality Engineering Magazine, Vol. 5, Issue 5 (Sep/Oct), pp. 16-22.

[4] Unit testing framework, http://msdn.microsoft.com/en-us/library/ms243147(v=VS.80).aspx.

[5] MichaelisMark. (2005)"A Unit Testing Walkthrough with Visual Studio Team Test, Microsoft®" MSDN (http://msdn.microsoft.com/en-us/library/ms379625(v=vs.80).aspx).

[6] NUnit home, http://nunit.org.

[7] JUnit home, http://www.junit.org.

[8] Kim Yong. (2003) "Efficient Use of Code Coverage in Large-Scale Software Development.", In the Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research CASCON '03, pp 145-155.

[9] Yi Wei, Yu Pei &Carlo AFuria (2008), "Is Coverage a Good Measure of Testing Effectiveness?", Chair of software engineering ETH Zurich, CH-8092 Zurich, Switzerland.

[10] Yang Qian, J. Jenny li &Wiess David (2007) "A survey of coverage based testing tools", Oxford university press on behalf of The British Computer Society.

[11] Piwowarski, Paul, Ohba, Mitsuru& Caruso, Joe (1993) "Coverage Measurement Experience during Function Test", Proc. 15th International Conference on software Engineering. IEEEComputer Society Press, pp 187-301, Baltimore, MD, USA.

[12] How to: Create a data-driven unit test, http://msdn.microsoft.com/en-us/library/ms182527.aspx

[13] TillmannNikolai &Halleux de Jonathan (2008),"White Box Test Generation for .NET", in Proc. of Tests and Proofs (TAP'08), Springer Verlag, pp 134-153.

[14] Parametrized unit testing with Microsoft® PEX, documentation, http://research.microsoft.com/en-us/projects/pex/documentation.aspx

[15] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, & Hong Mei (2010),"Test Generation via Dynamic Symbolic Execution for Mutation Testing", in Proceedings of the 26th IEEE International Conference on Software Maintenance ICSM 2010.

[16] King James (1976), "Symbolic execution and program testing", Communications of the ACM, Volume 19 Issue 7, July 1976, pp 385 – 394.

[17] Tao Xie, DarkoMarinov, Wolfram Schulte, & David Notkin (2005) "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution", In Proceedings, Lecture notes in Computer Science,TACAS'05 Springer, pp 365-381, Accessed at http://people.engr.ncsu.edu/txie/publications/tacas05.pdf.

[18] Configure Code Coverage (Visual Studio 2010) Using Test Settings in Visual Studio, http://msdn.microsoft.com/en-us/library/dd504821.aspx

[19] Exploring code with Microsoft® PEX, Tutorial for automated whitebox testing for .NET applications, PEX documentation, http://research.microsoft.com/en-us/projects/pex/digger.pdf

Authors

Mr Arpit Christi is currently working for Meridium services and labs Pvt.Ltd, Bangalore. He also teaches as visiting faculty at New HorizonCollege of engineering, Bangalore on a non-regular basis.He holds MS in computer science from California State University - Sacramento. His area of interests includes but not limited to software design, cloud computing, automated software testing and software performance engineering. He has extensive industry experience in software development, automated software testing and performance engineering.