Aiding the integration of automatically generated tests into pre-existing manually written test suites

Bryn Loftness

Colorado Mesa University

Research Advisor:

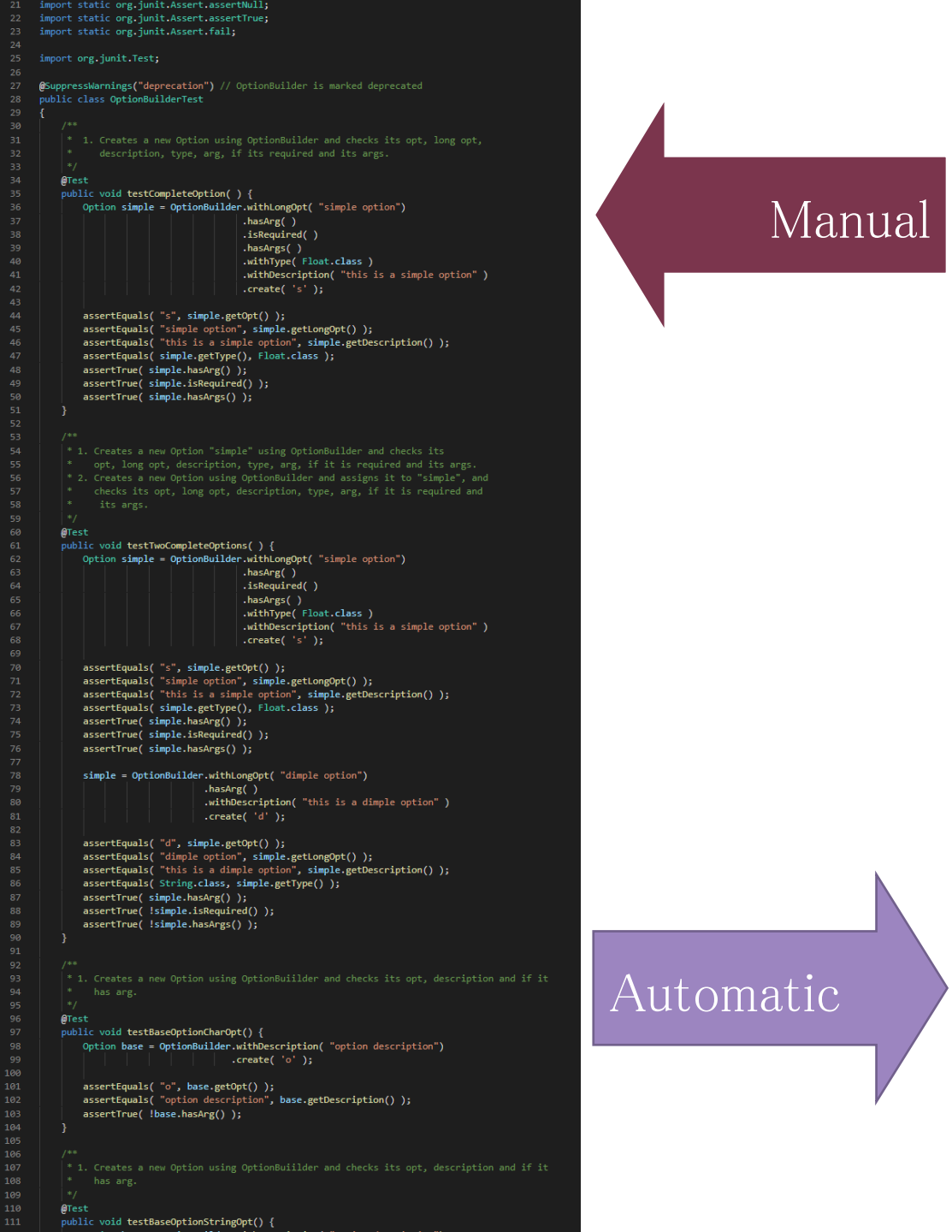
Dr. Venera Arnaoudova

Graduate Student Advisor:
Devjeet Roy

The Problem

Test Suite Reduction through the merging of redundant tests

Identifying optimal approach for selection of automatically generated tests for integration with similar manually written tests



```
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;

@SuppressWarnings("deprecation") // OptionBuilder is marked deprecated
public class OptionBuilderTest
{
    /**
     *  1. Creates a new Option using OptionBuilder and checks its opt, long opt,
     *     description, type, arg, if its required and its args.
     */
    @Test
    public void testCompleteOption( ) {
        Option simple = OptionBuilder.withLongOpt( "simple option")
                                     .hasArg( )
                                     .isRequired( )
                                     .hasArgs( )
                                     .withType( Float.class )
                                     .withDescription( "this is a simple option" )
                                     .create( 's' );

        assertEquals( "s", simple.getOpt() );
        assertEquals( "simple option", simple.getLongOpt() );
        assertEquals( "this is a simple option", simple.getDescription() );
        assertEquals( simple.getType(), Float.class );
        assertTrue( simple.hasArg() );
        assertTrue( simple.isRequired() );
        assertTrue( simple.hasArgs() );
    }

    /**
     * 1. Creates a new Option "simple" using OptionBuilder and checks its
     *    opt, long opt, description, type, arg, if it is required and its args.
     * 2. Creates a new Option using OptionBuilder and assigns it to "simple", and
     *    checks its opt, long opt, description, type, arg, if it is required and
     *     its args.
     */
    @Test
    public void testTwoCompleteOptions( ) {
        Option simple = OptionBuilder.withLongOpt( "simple option")
                                     .hasArg( )
                                     .isRequired( )
                                     .hasArgs( )
                                     .withType( Float.class )
                                     .withDescription( "this is a simple option" )
                                     .create( 's' );

        assertEquals( "s", simple.getOpt() );
        assertEquals( "simple option", simple.getLongOpt() );
        assertEquals( "this is a simple option", simple.getDescription() );
        assertEquals( simple.getType(), Float.class );
        assertTrue( simple.hasArg() );
        assertTrue( simple.isRequired() );
        assertTrue( simple.hasArgs() );

        simple = OptionBuilder.withLongOpt( "dimple option")
                              .hasArg( )
                              .withDescription( "this is a dimple option" )
                              .create( 'd' );

        assertEquals( "d", simple.getOpt() );
        assertEquals( "dimple option", simple.getLongOpt() );
        assertEquals( "this is a dimple option", simple.getDescription() );
        assertEquals( String.class, simple.getType() );
        assertTrue( simple.hasArg() );
        assertTrue( !simple.isRequired() );
        assertTrue( !simple.hasArgs() );
    }

    /**
     * 1. Creates a new Option using OptionBuiilder and checks its opt, description and if it
     *    has arg.
     */
    @Test
    public void testBaseOptionCharOpt() {
        Option base = OptionBuilder.withDescription( "option description")
                                   .create( 'o' );

        assertEquals( "o", base.getOpt() );
        assertEquals( "option description", base.getDescription() );
        assertTrue( !base.hasArg() );
    }

    /**
     * 1. Creates a new Option using OptionBuiilder and checks its opt, description and if it
     *    has arg.
     */
    @Test
    public void testBaseOptionStringOpt() {
```

Manual

Automatic

```
/*
 * This file was automatically generated by EvoSuite
 * Mon Sep 11 18:52:57 GMT 2017
 */

package org.apache.commons.cli;

import org.junit.Test;
import static org.junit.Assert.*;
import static org.evosuite.runtime.EvoAssertions.*;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET =
public class OptionBuilder_ESTest extends OptionBuilder_ESTest_scaffolding {

  /**
   * 1. Creates a new OptionBuilder using method withType of OptionBuilder
   *    and checks if it is not null.
   */
  @Test(timeout = 4000)
  public void test00()  throws Throwable  {
      Class<Object> class0 = Object.class;
      OptionBuilder optionBuilder0 = OptionBuilder.withType((Object) class0);
      assertNotNull(optionBuilder0);
  }

  /**
   * 1. Calls method "isRequired" of OptionBuilder.
   * 2. Creates a new OptionBuilder and checks if it is
   *    required and if its args are equal to -1.
   */
  @Test(timeout = 4000)
  public void test01()  throws Throwable  {
      OptionBuilder.isRequired();
      Option option0 = OptionBuilder.create("SQT$h");
      assertTrue(option0.isRequired());
      assertEquals((-1), option0.getArgs());
  }

  /**
   * 1. Calls method withLongOpt of OptionBuilder with argument "org.apache.commons.cli.OptionBuilder"
   * 2. Creates a new OptionBuilder using method create of OptionBuilder and checks if its args equals
   */
  @Test(timeout = 4000)
  public void test02()  throws Throwable  {
      OptionBuilder.withLongOpt("org.apache.commons.cli.OptionBuilder");
      Option option0 = OptionBuilder.create("");
      assertEquals((-1), option0.getArgs());
  }

  /**
   * 1. Calls method withArgName of OptionBuilder with argument "SQT$h"
   * 2. Creates a new OptionBuilder using method create of OptionBuilder and checks if its args equals
   */
  @Test(timeout = 4000)
  public void test03()  throws Throwable  {
      OptionBuilder.withArgName("SQT$h");
      Option option0 = OptionBuilder.create("SQT$h");
      assertEquals((-1), option0.getArgs());
  }

  /**
   * 1. Calls method hasOptionalArgs of OptionBuilder with argument 0
   * 2. Creates a new OptionBuilder using method create of OptionBuilder and checks if it has optiona
   *    and if its args equals 0.
   */
  @Test(timeout = 4000)
  public void test04()  throws Throwable  {
      OptionBuilder.hasOptionalArgs(0);
      Option option0 = OptionBuilder.create("");
      assertTrue(option0.hasOptionalArg());
      assertEquals(0, option0.getArgs());
  }

  /**
   * 1. Calls method hasArgs of OptionBuilder with argument 61
```

Our Approach

- Manual process is arduous and complex, even with test case scenarios and identifier renaming
- The systematic nature of our approach simulates a manual experts process of by-hand clustering:
  - Analyze jUnit manually written tests with Evosuite automatically written tests (in Java)
  - Identify key features of similarity
  - isolate and relate test cases based on the key features
  - Apply NLP-based information retrieval on key features to identify model(s) that best reduce false positive rates from the cluster list

Evaluation and Current Results

- We create an oracle based on results from manual clustering, we then evaluate model performance based on this oracle
- Clustering identifies groups of two or more test cases that are similar, our oracle simplifies these clusters into one-to-one matches
- Current approach narrows down potential one to one matches by over 50%

Identifying information that could help us categorize and find similar test cases…

```
/**
 *  1. Creates a new Option using OptionBuilder and checks its opt, long opt,
 *     description, type, arg, if its required and its args.
 */
@Test
public void testCompleteOption( ) {
    Option simple = OptionBuilder.withLongOpt( "simple option")
                                 .hasArg( )
                                 .isRequired( )
                                 .hasArgs( )
                                 .withType( Float.class )
                                 .withDescription( "this is a simple option" )
                                 .create( 's' );

    assertEquals( "s", simple.getOpt() );
    assertEquals( "simple option", simple.getLongOpt() );
    assertEquals( "this is a simple option", simple.getDescription() );
    assertEquals( simple.getType(), Float.class );
    assertTrue( simple.hasArg() );
    assertTrue( simple.isRequired() );
    assertTrue( simple.hasArgs() );
}
```

[1, creates, a, new, option, using, option, builder, and, checks, its, opt, long, opt, description, type, arg, if, its, required, and, its, args]

[public, void, test, complete, option, option, simple, option, builderwith, long, opt, simple, option, has, arg, is, required, has, args, with, type, floatclass, with, description, this, is, a, simple, option, create, s, assert, equals, s, simpleget, opt, assert, equals, simple, option, simpleget, long, opt, assert, equals, this, is, a, simple, option, simpleget, description, assert, equals, simpleget, type, floatclass, assert, true, simplehas, arg, assert, true, simpleis, required, assert, true, simplehas, args]

…Sequencing of words/keywords, similar words and keywords. And…

<?xml version="1.0" encoding="
<root>
<function>
<annotation>
@
<name>
Test
</name>
</annotation>
<type>
<specifier>
public
</specifier>
<name>
void
</name>
</type>
<name>
testCompleteOption
</name>
<parameter_list>
( )
</parameter_list>
<block>
{
<block_content>
<decl_stmt>
<decl>
<type>
<name>
Option
</name>
</type>
<name>
simple
</name>
<init>
=
<expr>
<call>
<name>
<name>
OptionBuilder
</name>
<operator>
.
</operator>
<name>
withLongOpt
</name>
</name>
<argument_list>
(
<argument>
<expr>
<literal type="string">
"simple option"
</literal>
</expr>
</argument>
)
</argument_list>
</call>

Digging a little deeper into sequencing by analyzing the Abstract Syntax Tree of the test case...

```
/**
 *  1. Creates a new Option using OptionBuilder and checks its opt, long opt,
 *     description, type, arg, if its required and its args.
 */
@Test
public void testCompleteOption( ) {
    Option simple = OptionBuilder.withLongOpt( "simple option")
                                 .hasArg( )
                                 .isRequired( )
                                 .hasArgs( )
                                 .withType( Float.class )
                                 .withDescription( "this is a simple option" )
                                 .create( 's' );

    assertEquals( "s", simple.getOpt() );
    assertEquals( "simple option", simple.getLongOpt() );
    assertEquals( "this is a simple option", simple.getDescription() );
    assertEquals( simple.getType(), Float.class );
    assertTrue( simple.hasArg() );
    assertTrue( simple.isRequired() );
    assertTrue( simple.hasArgs() );
}
```

[OptionBuilder, withLongOpt, hasArg, isRequired, hasArgs, withType, withDescription, create]

[assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt, assertEquals, literal, typestrings, simple, getOpt]

...Sequencing of words/keywords, similar words and keywords, similar methods being called, similar assert statements.

Our Progress So Far...

- Identified key features of test case and test case scenario to use as fruitful data inputs for similarity models (scenario words, assert statements, invoked methods)
- Series of models identifying test cases with these similar key features have reduced the potential cluster group to 50% of the possible cluster group, with plenty of room for more growth through further experimentation

Conclusions

Impacts of this Research:

Auto-identified clusters through a light-weight prototype can help programmers to select which test cases to keep, integrate, or discard

Next Steps:

Continue working towards reducing False Positive rates by finding/tuning models to identify key features of similarity (similar wording, similar methods being tested, similar sequences of actions, etc) and continue creating the ‘series of experts’ prototype

Acknowledgements

This material is based upon work supported by the National Science Foundation REU Program under Grant No. 1757632

Thank You!!
...Questions?