A Semiautomatic Test Case Generator Based on UML Activity Diagrams and Object Constraint Language

Tsung-Hsin Liu and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
{lth95m, naiwei}@cs.ccu.edu.tw

Abstract*- Software testing is the main activity to ensure the quality of software. This article applies the black-box approach to designing test cases. This article designs test cases based on the preconditions and postconditions in the Design by Contract software development approach. This article uses Unified Modeling Language (UML) activity diagrams and Object Constraint Language (OCL) to specify the preconditions and postconditions of Java methods. This article develops a semiautomatic tool to generate Java test classes for the JUnit framework based on UML activity diagrams and OCL.*

Keywords: Test case generation, Design by Contract, UML, activity diagrams, OCL.

1. Introduction

We usually do software testing for the following two reasons [6]. First, we need to evaluate the quality of software in order to ensure high quality software to customers. Second, we need to discover and remove the faults in software to improve the quality of software. Software testing is the main activity to ensure the quality of software.

Software testing is implemented by executing a suite of test cases and verifying if the software under test behaves correctly. Each test case is designed to test one of the functionalities of the software. A test case contains a test input and an expected output. The software is executed using the test input and the output of the software is verified with the expected output. If the two outputs are the same, the software is assumed to behave correctly for the functionality tested by this test case.

Software testing thus consists of test case design and test case execution. The JUnit framework is a popular test case execution tool for Java [3]. Given the test cases as a Java class and its methods, the JUnit framework can execute the test cases automatically. Test case design is a difficult job and there is no effective tool yet at this point. Hence, it is important to develop tools that can automatically or semi-automatically design test cases.

Test cases can be designed using either the black-box (or specification-based) approach or the white-box (or implementation-based) approach. These two approaches are complementary. This article applies the black-box approach to designing test cases. This article designs test cases based on the preconditions and postconditions in the Design by Contract software development approach [8]. We use Unified Modeling Language (UML) [10] activity diagrams and Object Constraint Language (OCL) [9] as a specification language to specify the preconditions and postconditions of Java methods.

This article develops a semiautomatic tool to generate Java test classes for the JUnit framework based on UML activity diagrams and OCL. This tool is developed in Eclipse development platform [5]. This tool is composed of four components: a diagram reader, a test path generator, a test path evaluator, and a test program generator, as shown in Figure 1. The diagram reader first reads in

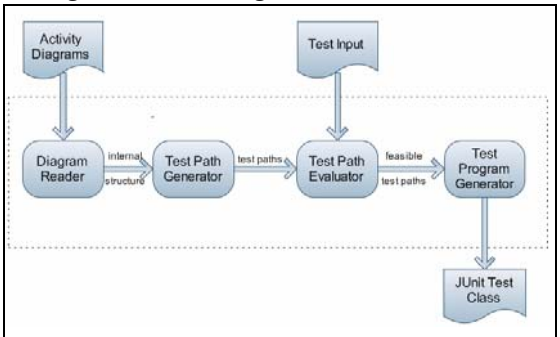


Figure 1. The system architecture of test case generator.

activity diagrams and OCL, and converts them into an internal data structure. The test path generator then enumerates all possible test paths on the activity diagrams. The test path evaluator allows the user to select test paths and displays the set of logical constraints required for traversing each selected test path. After given the test input for a selected test path, the test path evaluator can execute the test path and verify whether the test path is feasible. The test path evaluator can also display what test coverage criteria are satisfied by the set of selected feasible test paths. After the desired test coverage criterion is satisfied, the test program generator can generate the Java test classes for the JUnit framework according to the set of selected feasible test paths.

The rest of this article is organized as follows. Section two introduces the related work. Section three gives a brief introduction to the specification language: UML activity diagrams and OCL. Sections four to seven describe each of the four components of our semiautomatic test case generator, respectively. Finally, Section eight concludes this article.

2. Related Work

The most related work is the Java Modeling Language (JML). JML is a behavioral interface specification language [7]. JML uses Java syntax and specifies the preconditions and postconditions for each Java module. A user first writes the specification of a Java module using JML. The user then uses the JML compiler to compile the specification, and uses the JML runtime assertion checker to verify the implementation with the specification [4]. JML is specific to Java. Since UML activity diagrams and OCL are language independent, our work can be easily extended to other languages.

3. The Specification Language

This section briefly introduces our specification language: UML activity diagrams and OCL, and describes a running example for this article.

The input and output relationships of a Java method can be classified as a set of equivalence classes. The input and output relationships in the same equivalence class satisfy the same set of logical constraints on the input. The relations of logical constraints are usually defined using a decision table. This article uses UML activity diagrams as a graphical notation for decision tables.

UML activity diagrams only describe the logical structure among logical constraints. The explicit representation of logical constraints and the input and output relationships are specified using OCL expressions as preconditions and postconditions. With these OCL expressions and given input test data, we can then compute the expected output by evaluating these OCL expressions.

The following two subsections describe the subset of UML activity diagrams and OCL expressions that we can handle.

3.1. UML activity diagrams

UML activity diagrams are used to describe the logical structure of logical constraints in a Java method. Each Java method may have some parameters acting as input data to the method. Each Java method may have some parameters or return value acting as output data from the method. We will use input variables and output variables to refer to these parameters and return value. We may also use some temporary variables to define the intermediate values of OCL expressions. These temporary variables can only be defined once.

We use the following five types of nodes in an activity diagram: the start node, the end node, action nodes, decision nodes, and note nodes. The start node represents the entry of the diagram, and the end node represents the exit of the diagram.

An action node represents a relationship among variables. These relationships are defined using OCL expressions. They are usually used to define temporary variables or output variables. A decision node represents a logical constraint C. A decision node has two outgoing edges. Each edge is associated with a guard condition corresponding to one of the two OCL expressions C and ¬C. A note node represents a set of preconditions or postconditions. A note node usually is associated with the start node, the end node, or an action node.

3.2 Object constraints language

OCL is a formal language for describing expressions on UML models defined in UML [9]. OCL is used to define variables, logical constraints, preconditions and postconditions.

We classify the variables into input variables (IN), output variables (OUT), and temporary variables (TMP), to represent data passed into, passed back from, or used locally inside a method.

These variables are declared in a note node associated with the start node.

Preconditions define what input conditions must be satisfied before calling the method. Preconditions are also defined in a note node associated with the start node.

Postconditions define what output conditions must be satisfied after calling the method under the assumption that input conditions are satisfied. Postconditions are usually defined in a note node associated with the end node or an action node.

Some local preconditions and postconditions are possible. These local preconditions and postconditions are defined in a note node associated with an action node.

Logical constraints are associated with outgoing edges of decision nodes.

A complete path from the start node to the end node represents an equivalent class of the input and output relationships of a Java method. The set of complete paths consists of the maximal set of test cases.

3.3 An example

We illustrate the specification of a running example using UML activity diagrams and OCL in Figure 2. The example is the triangle problem: given the lengths of the three edges of a triangle, the Java method returns the type of the triangle. The types of the triangle may be one of NoTaTriangle, Scalene, Isosceles, or Equilateral.

4. Diagram Reader

The diagram reader reads in UML activity diagrams and OCL, and converts them into an internal data structure. We draw UML activity diagrams using Omondo EclipseUML tools [11]. The diagrams are saved in XML files. We use the Apache Xerces DOMParser to read and parse the XML files [1]. We then convert them into an internal data structure that can be processed more efficiently.

5. Test Path Generator

The test path generator can enumerate all possible test paths on the activity diagrams. Figure 3 shows the result after reading in the specification of the triangle problem and enumerating all the possible test paths of the triangle problem. There are eight possible test paths in the activity diagram of the triangle problem. The test path generator uses the depth-first traversal algorithm to enumerate all possible test paths on the activity diagrams.

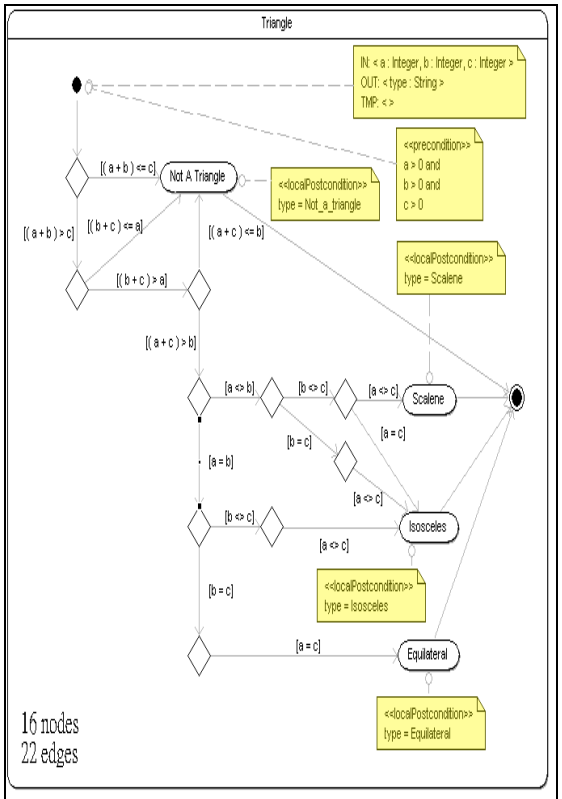


Figure 2. The specification for the triangle problem.

Note that some test paths in an activity diagram may be infeasible. A test path is infeasible if no input data can satisfy all the logical constraints on the test path; otherwise, it is feasible. We need to depend on the user and the test path evaluator to verify feasible test paths.

6. Test Path Evaluator

The test path evaluator allows the user to select a test path and then displays the set of logical constraints required for traversing the selected test path. Figure 4 shows the result of selecting the fourth test path. This test path contains the set of logical constraints that ensures an equilateral triangle.

After looking at the set of logical constraints, the user may use some constraint solving tool to solve this set of logic constraints. A Constraint Logic Programming system like ECLiPSe may be such a tool [1]. For the fourth test path, we can easily provide a solution, such as {a = 3, b = 3, c = 3}.

Given one of the solutions of the set of logical

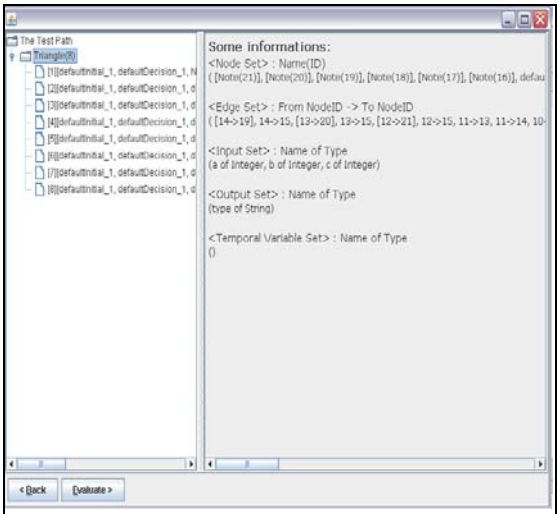


Figure 3. The set of test paths for the triangle problem.

constraints as the test input, the test path evaluator can execute the test path and verify whether all the logical constraints on the test path is satisfied. If it is, this test path is feasible and the evaluated output can be regarded as an expected output. The test path, the test input, and the expected output can then be saved as a test case. Figure 5 shows that given the test data, the expected output is "Equilateral", and the fourth test path is feasible.

The test path evaluator can also display what test coverage criteria are satisfied by the set of selected feasible test paths. The test path evaluator can verify all-node, all-edge, and all-path coverage criteria. The test path evaluator can also display what percentage of a test coverage criterion has been satisfied by the set of selected feasible test

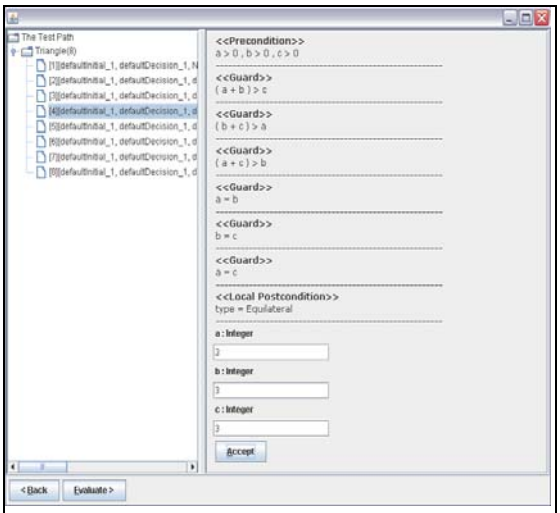


Figure 4. The logical constraints and test input for the fourth test path of the triangle problem.

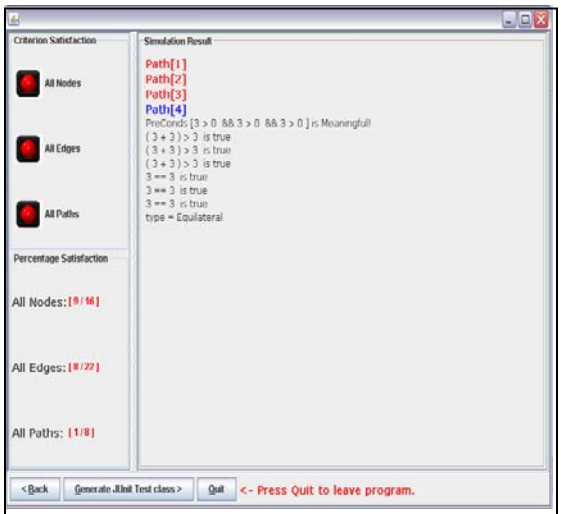


Figure 5. The expected output and coverage criteria satisfaction rates for fourth test path of the triangle problem.

paths. Figure 5 shows that if only the fourth test path is selected, then 9/16 all-node coverage criterion, 8/22 all-edge coverage criterion, and 1/8 all-path coverage criterion, have been satisfied.

The execution of a test path is achieved by implementing an interpreter for OCL expressions. This interpreter first interprets the declarations of variables. It then reads in the input given by the user. The interpreter then checks whether all the preconditions are satisfied. If any of the preconditions is not satisfied, the test path is infeasible and the interpreter terminates.

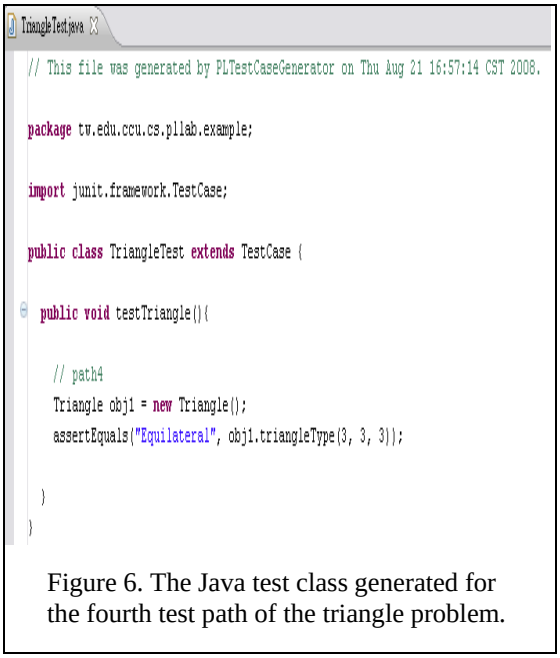
During the traversing of the test path, the interpreter checks whether the corresponding logical constraint is satisfied if a decision node and its outgoing edge is traversed. If any of the preconditions is not satisfied, the test path is infeasible and the interpreter terminates.

It evaluates the local postconditions or postconditions if an action node and its associated note nodes are traversed. The interpreter finally evaluates the postconditions if the end node and its associated note nodes are traversed. If the interpreter successfully traverses the end node, the test path is feasible.

7. Test Program Generator

After the desired test coverage criterion is satisfied, the test program generator can generate a Java test class for the JUnit framework according to the set of selected feasible test paths.

For each selected feasible test path, the test program generator first generates code to create an object of the class in which the method under test



```
// This file was generated by PLTestCaseGenerator on Thu Aug 21 16:57:14 CST 2008.

package tw.edu.ccu.cs.pllab.example;

import junit.framework.TestCase;

public class TriangleTest extends TestCase {

  public void testTriangle(){

    // path4
    Triangle obj1 = new Triangle();
    assertEquals("Equilateral", obj1.triangleType(3, 3, 3));

  }
}
```

Figure 6. The Java test class generated for the fourth test path of the triangle problem.

is. The test program generator then generates code to invoke the Java method under test with test data as parameters. The test program generator finally generates code to assert that the return value of the Java method under test is the same as the expected output computed by the test path evaluator. The Java class generated by the test program generator with only the fourth test path selected is shown in Figure 6. With such a Java test class, the user can test the Java method using the JUnit tool.

8. Conclusion

This article has applied the black-box approach to semiautomatically design test cases. The approach to designing test cases is based on the preconditions and postconditions in the Design by Contract software development approach. This article uses Unified Modeling Language activity diagrams and Object Constraint Language as a specification language to specify the preconditions and postconditions of Java methods. This article has also described a tool that semiautomatically generates Java test classes for the JUnit framework.

The tool still needs to depend on the user to determine the test data for a selected test path. We are investigating the means of integrating the Constraint Logic Programming system into our tool so that we can also automate the determination of the test data for a selected test path.

References

[1] Apache, *Xerces DOMParser*, http://www.docjar.com/.

[2] K. R. Apt and M. Wallace, *Constraint Logic Programming Using ECLiPSe*, Cambridge University press, 2007.

[3] M. Beck and E. Gamma, *JUnit CookBook*, http://junit.org/.

[4] Y. Cheon and G.. Leavens, "A Runtime Assertion Checker for the Java Modeling Language," In *Proceedings of International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada. CSREA Press, June 2002, pp. 322-328.

[5] Eclipse Foundation, http://www.eclipse.org/.

[6] P. C. Jorgensen, *Software Testing: A Craftman's Approach*, CRC press, 1995.

[7] G.. T. Leavens, A. L. Baker, and C. Ruby, "JML: A Notation for Detailed Design," In *Behavioral Specifications of Businesses and Systems*, editors, H. Kilov, B. Rumpe, and I. Simmonds, Chapter 12, Kluwer, 1999, pp. 175-188.

[8] B. Meyer, "Design by Contract," In *Advances in Object-Oriented Software Engineering*, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, pp. 1-50.

[9] Object Management Group, *Object Constraint Language Specification*, *Version 2.0*, http://www.omg.org/, 2006.

[10] Object Management Group, *UML 2.1.1 Specification*, http://www.uml.org/, 2007.

[11] Omondo, *EclipseUML*, http://www.eclipsedownload.com/.