奠基於物件限制語言與限制邏輯程式的自動測試工具
An Automatic Testing Tool Based on Object Constraint Language and Constraint Logic Programming

胡伊婷　林迺衛
國立中正大學　資訊工程學系
Yi-Tin Hu and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi 621, Taiwan, R.O.C.
Email: {hyt96m,naiwei}@cs.ccu.edu.tw

Abstract

Software testing is the main activity to ensure the quality of software. This article applies a black-box testing technique to automatically generate test cases. This article generates test cases based on the preconditions and postconditions in the Design by Contract software development approach. This article uses Object Constraint Language to specify the preconditions and postconditions of Java methods. This article develops an automatic testing tool to generate Java test classes for the Java classes under test. The testing framework is based on the JUnit framework. The automatic generation of test input and expected output for each test case is based on the powerful constraint solving capability of constraint logic programming. This approach can generate the test input and expected output simultaneously.

1. Introduction

According to 2006 CHAOS report of the Standish Group, high quality software is still very hard to achieve at present [7]. The *software testing* activity still remains one of the main activities to assure software quality. The cost of the software testing activity usually takes about half of the cost of the entire software process. This high cost of the software testing activity is the main reason to hinder the achievement of high quality software.

Many software testing techniques have been developed during the last several decades. However, most of these software testing techniques are still performed *manually* at present. This is why the cost of the software testing activity is so high. The *automation* of the software testing activity should be able to significantly reduce the cost of the software testing activity.

The software testing activity consists of the *design* of test cases and the *execution* of test cases. There are two types of techniques for designing test cases: *black-box testing* (or *specification-based testing*) and *white-box testing* (or *program-based testing*) [3]. The black-box testing techniques are based on the *functional specification* of a software system and focus on the coverage of the specified *external* behaviors of the software system. The white-box testing techniques are based on the *source code* and focus on the coverage of the *internal* structure of the source code. These two types of testing techniques are complementary. The white-box testing techniques alone may fail to reveal that some portions of the specification are missed in the source code. On the other hand, the black-box testing techniques alone may fail to reveal that some portions of the source code are not contained in the specification. Therefore, employing both types of testing techniques is necessary to assure high quality software.

Design by Contract is a systematic approach for developing bug free software [4]. In Design by Contract approach, the functional behaviors of a class are specified as a contract. The contract specifies the correct interactions between a class and its clients as a set of obligations that need to be fulfilled by both parties. If both parties fulfill their obligations, the interactions between the two parties are guaranteed to be correct.

Developing the contract of a software system is the first and perhaps the most difficult task of developing the software system. However, if we cannot develop the contract of a software system, it is little likelihood that the developed software system will behave as it is supposed to behave.

In Design by Contract approach, three kinds of constraints are used to specify the functional behaviors of a class. The *preconditions* of a method in the class specify the set of constraints that need to be satisfied before the method can correctly execute. The *postconditions* of a method in the class specify the set of constraints that is guaranteed to be satisfied after the method is correctly executed. The *invariants* of a class specify the set of constraints that is always satisfied by an object of the class.

Object Constraint Language (OCL) is a specification language that can be used to specify the three kinds of constraints in Design by Contract approach [5]. Given the contract for a Java class

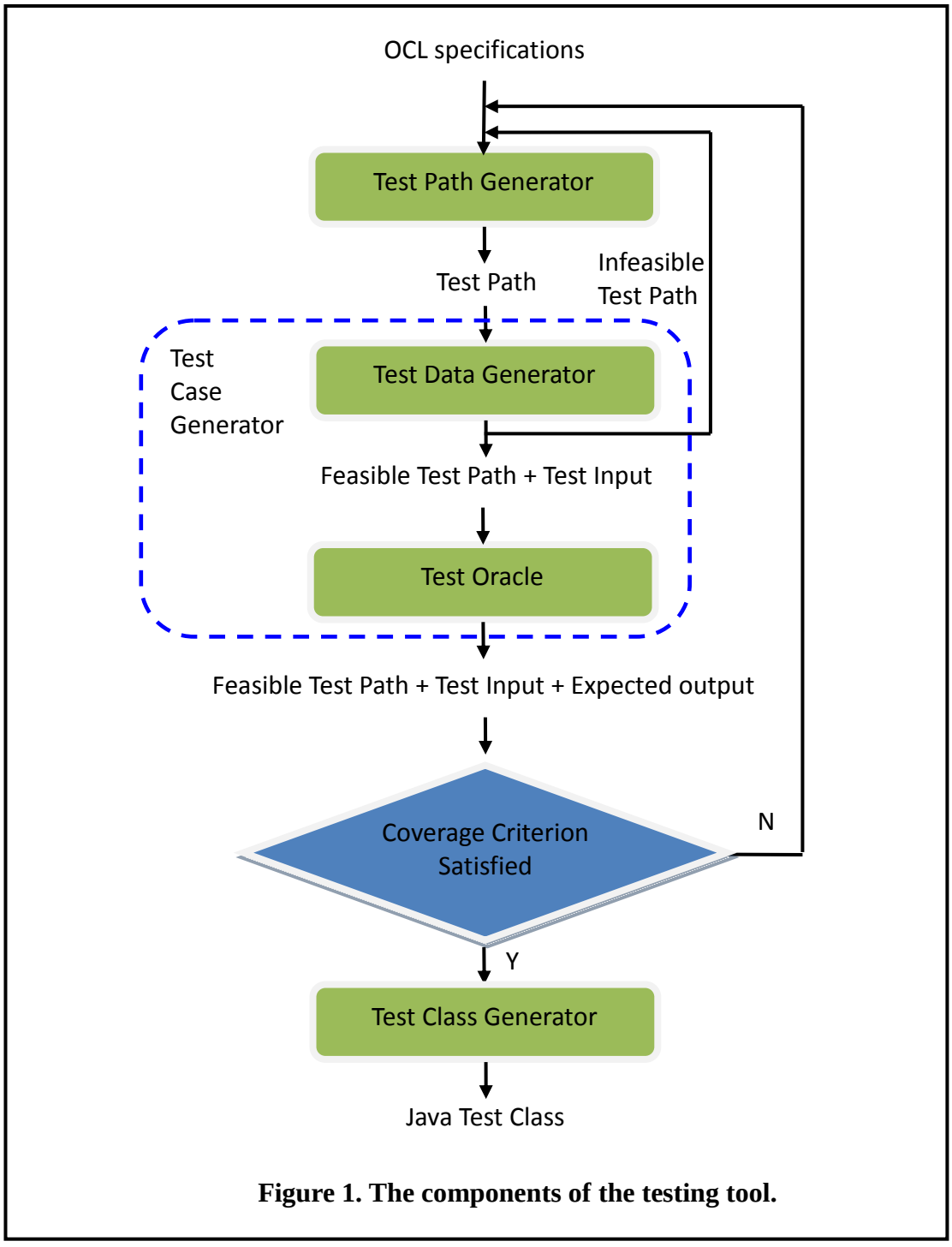

Figure 1. The components of the testing tool.

specified in OCL, the test cases for the methods in the class can be designed using the black-box testing technique. Each test case consists of a pair of input and expected output.

In black-box testing, test cases are generated repeatedly in the following three steps, as shown in Figure 1. First, a test (or execution) path is generated from the specification. Each test path covers a sub-domain of the input. This step is implemented as a *test path generator.* Second, the feasibility of the test path is verified by solving the set of logical predicates appearing on the test path. If the set of logical predicates is satisfiable, this test path is feasible. In this case, a solution of the set of logical predicates is used as a representative input from the sub-domain covered by this test path. Otherwise, the test path is infeasible. This step is implemented as a *test data generator*. Third, the expected output with respect to the input is determined. This step is implemented as a *test oracle*. These three steps are repeated until the set of generated test cases satisfies a specific test coverage criterion. Most modern black-box testing tools focus on the automation of test path generators. The tasks of test data generators

and test oracles are still usually performed manually by programmers or testers.

This paper proposes an approach that can automatically generate test input and expected output simultaneously based on the powerful constraint solving capability of Constraint Logic Programming systems (CLP). This approach allows us to design an automatic testing tool that consists of a test path generator, a test case generator, and a test class generator, as shown in Figure 1. The test case generator automatically generates test input and expected output for each feasible test paths. The test class generator automatically generates a Java test class for the tested Java class. This paper uses the Java development environment Eclipse to develop the testing tool [6]. This paper uses the Constraint Logic Programming system ECLiPSe to solve the constraints [1].

The rest of the paper is organized as follows. Section 2 describes a running example for this paper. Section 3 gives a contract for the running example using OCL. Section 4 describes the test path generator. Section 5 describes the test case generator. Section 6 describes the test class generator. Finally, Section 7 concludes this paper.

2. The Running Example

We now describe the running example that is used to illustrate the automatic test case generation for Java methods. Consider the following Java class Triangle.

```
class Triangle {
  Triangle(int sa, int sb, int sc);
  public String category();
  private int a, b, c;
}
```

Each of the objects of the class Triangle represents a triangle. The variables a, b, and c represents the lengths of the three sides of the triangle. The method category() returns the category of a Triangle object determined by the lengths of the three sides: “Equilateral”, “Isosceles”, “Scalene”, or “NotATriangle”. A Triangle object is “NotATriangle” if it does not satisfy any of the following three constraints: {a + b > c, a + c > b, b + c > a}. Otherwise, it is a triangle. A Triangle object is an “Equilateral” triangle if it satisfies the following two constraints: {a = b, a = c}. A Triangle object is an “Isosceles” triangle if it satisfies one and only one of the following three constraints: {a = b, a = c, b = c}. Otherwise, It is a “Scalene” triangle.

3. The Specification

OCL is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects. The OCL expression for the method category() in the running example can be given as follows:

```
context Triangle::category () : String
pre: true
post: result =
if (a + b <= c) then
  “NotATriangle”
else if (a + c <= b) then
  “NotATriangle”
else if (b + c <= c) then
  “NotATriangle”
else if (a = b and a = c) then
  “Equilateral”
else if (a = b) then
  “Isosceles”
else if (a = c) then
  “Isosceles”
else if (b = c) then
  “Isosceles”
else
  “Scalene”
endif
```

The expression true means that there is no precondition for the method category(). The if-then-else-endif expression specifies the postcondition for the method category().

4. Test Path Generator

The path generator is basically an OCL translator that parses the OCL specifications of a Java method and converts it into a control flow graph. It can then enumerate all the test paths in the control flow graph.

Several enumeration schemes are possible. If the testing resource is limited so that only portion of test paths can be tested and the usage profile for every test path is available, then a portion of test paths that is most cost-effective can be enumerated. Otherwise, a depth-first traversal of the control flow graph suffices to enumerate all the test paths.

For the running example, there are a total of 8 test paths. Three test paths for “NotATriangle” triangle objects. One test path for “Equilateral” triangle objects. Three test paths for “Isosceles” triangle objects. One test path for “Scalene” triangle objects.

5. Test Case Generator

The test case generator performs two tasks. First, the test case generator converts each test path of the control flow graph into an ECLiPSe predicate. This predicate specifies the set of constraints on the test path. A query for this predicate can be used to

generate test input and expected output simultaneously.

For the running example, the set of constraints on the test path for "Equilateral" triangle objects is as follows:

```
{a + b > c, a + c > b, b + c > a, a = b, a = c}.
```

Then, the test case generator will convert this test path into the following ECLiPSe predicate:

```
%include
:- lib(ic).

categoryTriangle(A, B, C, Result) :-
% Domains
   [A, B, C] :: -32768 .. 32767,
% Constraints
   A + B #> C, A + C #> B, B + C #> A,
   A #= B, A #= C,
   Result = "Equilateral",
% Solving
   indomain(A),
   indomain(B),
   indomain(C),
   locate([A, B, C], 0.1).
```

Query the ECLiPSe system using the following query:

```
categoryTriangle(A, B, C, Result).
```

The ECLiPSe system will return

```
A = 1
B = 1
C = 1
Result = "Equilateral"
```

That is, the ECLiPSe system will return both the test input: (1, 1, 1) and the expected output: "Equilateral" simultaneously.

If there is no solution for such a query, then the test path corresponding to this predicate is an infeasible test path. The test case generator will generate the test input and expected output for all feasible test paths that satisfies the test coverage criterion.

Second, the test case generator converts the entire control flow graph for a method *m* into an ECLiPSe predicate *p*. This predicate *p* can be used to generate automatically the expected output if *m* is a recursive method. This predicate *p* can also be used later to generate automatically the expected output for a method *m*' if *m*' calls *m*.

6. Test Class Generator

The test class generator generates a Java test class for the class under test. The test class is based on the JUnit framework []. For each Java method under test, there is a corresponding test method.

For each selected feasible test path, the test class generator first generates code to create an object of the class. It then generates code to invoke the method with the test input corresponding to this test path. Finally, it generates code to check if the actual output of the invoked method is the same as the expected output.

For the running example, the test class generator generates the following test method for the method category().

```
public class TriangleTest extends TestCase
{
   public void testTriangle()
   {
   // path 1
   Triangle o1 = new Triangle(-32768,-32768,1);
   assertEquals("NotATriangle", o1.category());

   // path 2
   Triangle o2 = new Triangle(-32768,1,-32768);
   assertEquals("NotATriangle", o2.category());

   // path 3
   Triangle o3 = new Triangle(1,-32768,-32768);
   assertEquals("NotATriangle", o3.category());

   // path 4
   Triangle o4 = new Triangle(1,1,1);
   assertEquals("Equilateral", o4.category());

   // path 5
   Triangle o5 = new Triangle(2,2,1);
   assertEquals("Isosceles", o5.category());

   // path 6
   Triangle o6 = new Triangle(2,1,2);
   assertEquals("Isosceles", o6.category());

   // path 7
   Triangle o7 = new Triangle(1,2,2);
   assertEquals("Isosceles", o7.category());

   // path 8
   Triangle o8 = new Triangle(2,3,4);
   assertEquals("Scalene", o8.category());
   }
}
```

indomain(C), locate([A, B, C], 0.1). indomain(B), indomain(C),

7. Conclusion

Automatic generation of input data and expected output is the most difficult tasks in software testing activity. The automation of any task depends on the formal specification of the task. This article uses Object Constraint Language as a formal

language to specify the preconditions and postconditions of Java methods. This article uses the powerful constraint solving capability of constraint logic programming to automatically generate input data and expected output for test cases. This approach can generate the test input and expected output simultaneously. This article shows that the integration of OCL and CLP makes the design and implementation of an automatic testing tool for Java methods possible.

References

[1] K. R. Apt and M. G. Wallace, *Constraint Logic Programming Using ECLiPSe*, Cambridge University Press, 2007.

[2] K. Beck and E. Gamma, *JUnit Cookbook*, http://junit.sourceforge.net/.

[3] B. Bezier, *Software Testing Techniques*, 2nd Edition, Van Nostrand, 1990.

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

[5] Object Management Group, *Object Constraint Language Specification*, Version 2.0, 2006.

[6] Object Technology International Incorporation, *Eclipse Platform Technical Overview*, 2003.

[7] Standish Group, *2006 CHAOS Report*, 2007.