Automatic Black-Box Method-Level Test Case Generation Based on Constraint Logic Programming

Yi-Tin Hu and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi 621, Taiwan, R.O.C.
e-mail: {hyt96m,naiwei}@cs.ccu.edu.tw

Abstract—**Software testing is the main activity to ensure the quality of software. This article proposes a testing framework that applies a black-box testing technique to automatically generate test cases for Java methods. The specification of Java methods is based on the Design by Contract software development approach. This article uses UML class diagrams and the Object Constraint Language to describe the specification of Java methods. The automatic generation of test cases is based on the unification mechanism and the powerful constraint solving mechanism of constraint logic programming. A novel characteristic of this approach is that the test input and its corresponding expected output of a test case are simultaneously generated in a unified fashion.**

Keywords-black-box testing; automatic test case generation; constraint logic programming

I. INTRODUCTION

According to 2006 CHAOS report of the Standish Group, high quality software is still very hard to achieve at present [15]. 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. 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: the *black-box* (or *specification-based*) technique and the *white-box* (or *program-based*) technique [3]. The black-box technique is based on the functional specification of a software system and focuses on the coverage of the specified external behaviors of the software system. The white-box technique is based on the source code and focuses on the coverage of the internal structure of the source code. These two types of techniques are complementary. The white-box technique 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 technique alone may fail to reveal that some portions of the source code are not contained in the specification. Therefore, employing both types of techniques is necessary to assure high quality software. This article is based on the black-box technique.

The Design by Contract software development approach is a systematic approach for developing bug free software [8]. In the Design by Contract approach, the functional behaviors (or specifications) 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 difficult to develop a software system that will behave as it is supposed to behave.

In the Design by Contract approach, three kinds of *constraints* are used to specify the functional behaviors of a class. The *invariants* of a class specify the set of constraints that is satisfied by any object of the class at any time. 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 Unified Modeling Language (UML) is a standardized general-purpose modeling language. A UML class diagram (CD) describes the attributes and the methods in a Java class. The Object Constraint Language (OCL) is a specification language that can be used to specify the three kinds of constraints in the Design by Contract approach [11] for a class diagram. Given the contract for a Java class specified in OCL, the test cases for the methods in the class can be automatically designed using the black-box technique.

978-1-4244-7638-1/10/$26.00 ©2010 IEEE

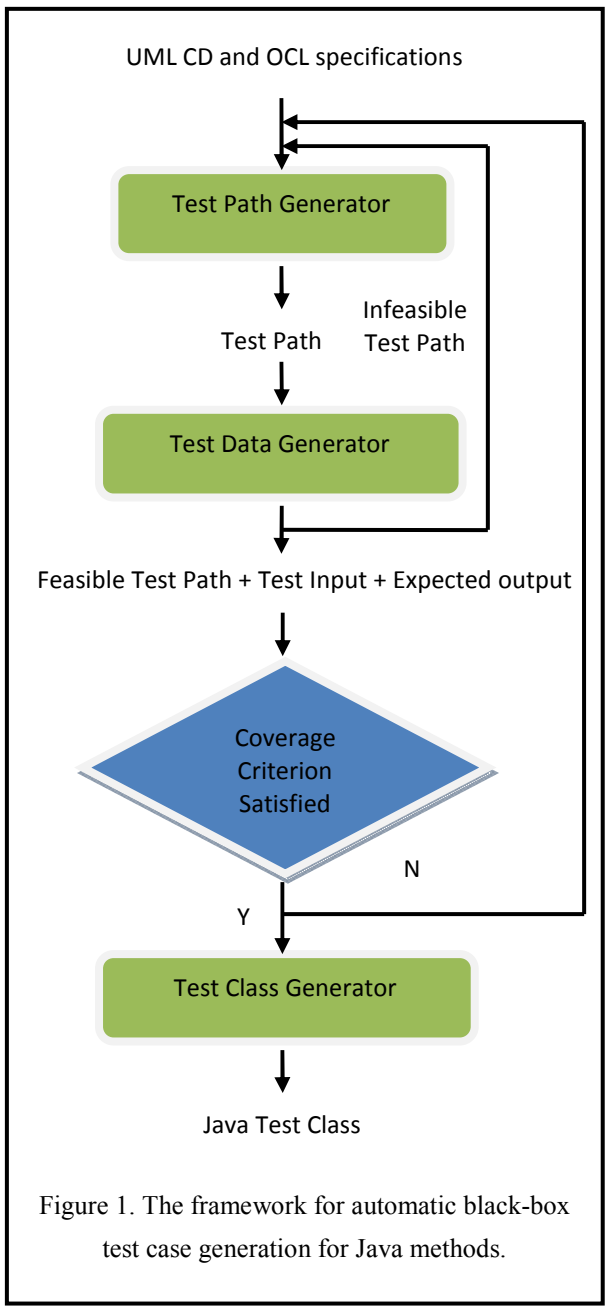

Figure 1. The framework for automatic black-box test case generation for Java methods.

The framework for automatic black-box test case generation for Java methods is shown in Figure 1. This framework contains three components: a test path generator, a test data generator, and a test class generator. The test path generator first reads in a UML class diagram and the corresponding OCL specifications for the methods in the class. It then enumerates a possible test path from the specification. The test data generator then checks the feasibility of the test path by solving the set of logical constraints appearing on the test path. If the set of logical constraints is satisfiable, this test path is feasible. In this case, a solution of the set of logical constraints is used as a representative test input and its corresponding expected output. Otherwise, the test path is infeasible and is abandoned. The test path generator and the test data generator perform repeatedly until the set of generated feasible test paths satisfies a specific test coverage criterion. The test class generator then generates a Java test class for the tested Java class based on the set of generated test inputs and their corresponding expected outputs.

Most modern black-box unit testing tools focus on the automation of test path generators and the test class generator. The tasks of test data generators are still usually performed manually by programmers or testers. This article proposes an approach that can automatically generate the test input and expected output simultaneously based on the powerful constraint solving capability of the Constraint Logic Programming system ECLiPSe [1].

The automatic execution of test cases is based on the JUnit framework [2]. This article uses the Java development environment Eclipse [12] to develop this framework.

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

II. A Running Example

We now describe a 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 constructor Triangle(sa, sb, sc) creates a Triangle object with lengths sa, sb, and sc. The constructor Triangle() succeeds only if its three arguments satisfy the following three constraints: {sa + sb > sc, sa + sc > sb, sb + sc > sa}. Otherwise, an exception is raised. The method category() returns the category of a Triangle object based on the lengths of its three sides: "Equilateral", "Isosceles", or "Scalene". 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.

The OCL specification for the invariant of the objects of this class is given as follows:

```
context Triangle
inv: a + b > c and a + c > b and b + c > a
```

This invariant specifies that these three constraints must be satisfied for every triangle object at any time.

The OCL specification for the method category() is given as follows:

```
context Triangle::category () : String
```

```
pre: true
post:
   if (a@pre = b@pre) then
     if (a@pre = c@pre) then
        result = "Equilateral"
     else
        result = "Isosceles"
     endif
   else
     if (a@pre = c@pre) then
       result = "Isosceles"
     else
       if (b@pre = c@pre) then
         result = "Isosceles"
         else
            result = "Scalene"
         endif
       endif
   endif
```

The precondition true for the method category() specifies that no constraint is needed to be satisfied in order to invoke this method. The postcondition for the method category() specifies the value returned by this method for the three categories of triangles, respectively. The expression a denotes the value of attribute a right after the method invocation, while the expression a@pre denotes the value of attribute a right before the method invocation.

III. THE TEST PATH GENERATOR

The test path generator consists of three steps. First, it reads in and parses a UML class diagram and its corresponding OCL specifications. Second, for each method in a class, it then constructs a control flow graph for its precondiction and a control flow graph for its postcondition. Third, it then enumerates all the possible test paths in these two control flow graphs.

The enumeration schemes of test paths depend on the test coverage criterion chosen. The test coverage criterion can be control-flow based: all-decision, all-condition, or all-decision-condition, or data-flow based: all-define, all-use, or all-define-use. In this article, we will assume that the all-decision (or all-branch) test coverage criterion is used.

For the running example, there is no control flow graph for the precondition of the method category(). The control flow graph for the first postcondition of the method category() is given in Figure 2. There are a total of 5 possible test paths for this control flow graph. One test path corresponds to "Equilateral" triangle objects, three test paths correspond to "Isosceles" triangle objects, and one test path corresponds to "Scalene" triangle objects. Using the all-decision test coverage criterion, these five paths will be enumerated by the test path generator and passed to the test data generator to generate the test input and expected output for each of these paths.

IV. THE TEST DATA GENERATO

The test data generator generates test input and expected output for each test path passed from the test path generator.

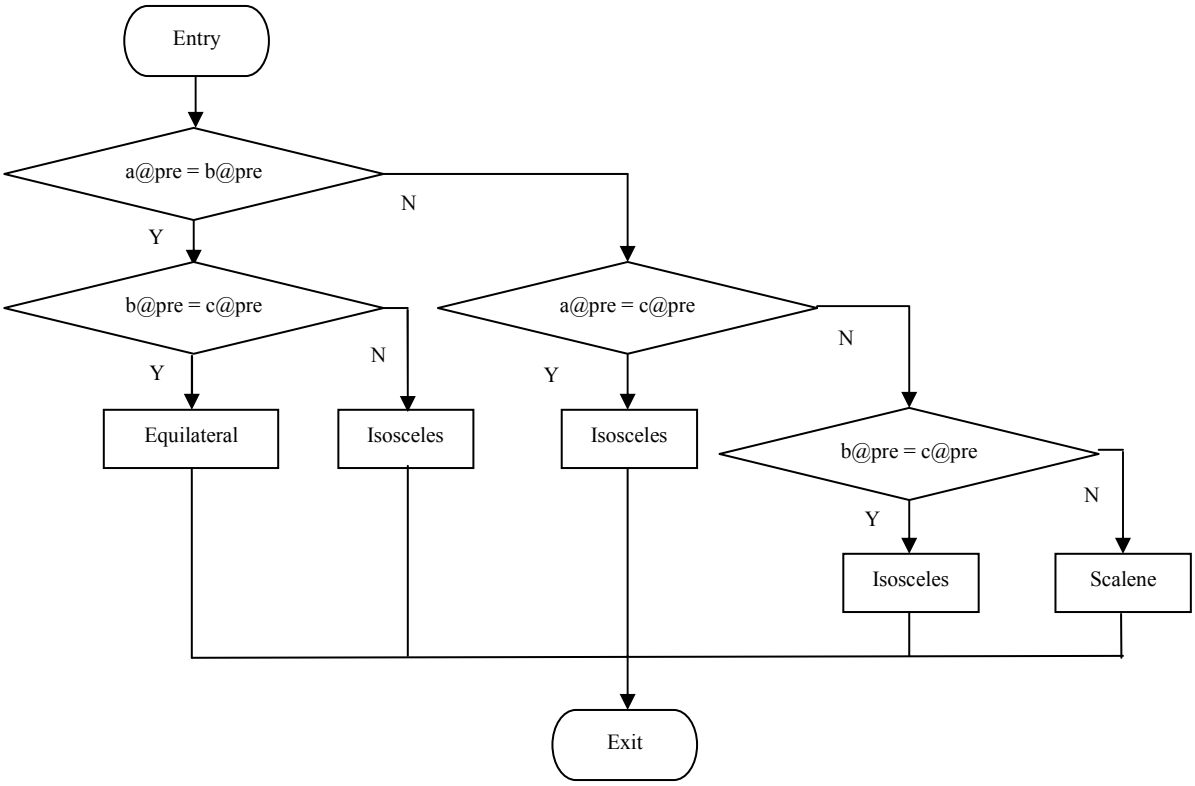

Figure 2. The control flow graph for the first postcondition of the method categort().

The test input and expected output are generated by solving the set of constraints on the test path. This set of constraints is first transformed into ECLiPSe predicates and these predicate are then solved by the powerful constraint solving capability of the ECLiPSe system.

The powerful constraint solving capability of the ECLiPSe system can be illustrated by the following example.

```
%Include constraint solving library
:- lib(ic).
foo(A, B, C) :-

% Domains of variables
   [A, B, C] :: 1 .. 3,
% Constraints on variables
   A #= B + C,
% Solving constraints
   indomain(A),
   indomain(B),
   indomain(C),
   locate([A, B, C], 0.1).
```

where ic is the library for solving constraints on finite domains. The ECLiPSe system supports several other libraries. Based on the unification mechanism of logic programming, the predicate foo/3 can be queried with any number of variable arguments. For example, the predicate foo/3 can be queried using any of foo(3, 1, 2), foo(3, 1, C), foo(3, B, C), foo(A, B, C), and so on. This unification mechanism allows us to solve the test input and expected output simultaneously. By specifying the domains of variables, the constraint solving library can find all the solutions satisfying the constraints in the predicate. For the example, the ECLiPSe system will returns

```
A = 3,
B = 1,
C = 2,
```

and

```
A = 3,
B = 2,
C = 1,
```

as solutions for the query foo(A, B, C).

For each method method in the class class, the test data generator will convert the control flow graph for the return value of the method into two ECLiPSe predicates classMethod/5 and testClassMethod/5.

Each clause of the predicate classMethod/5 specifies the set of constraints on a distinct test path. This predicate has an argument representing the arguments of the method and an argument representing the return value of the method. Just like each Java method has an implicit extra argument that passes the calling object itself to the method so that the method can access the attributes of the calling object, this predicate also has an argument that represents the state of the object right before the method invocation. In addition, this predicate also has an argument that represents the state of the object right after the method invocation.

For the running example, the test data generator will generate a predicate triangleCategory(PathNo, OState, Arguments, Result, NState). The argument PathNo is a list of identification number of test paths with the first element being a test path for the method category() and the rest of the elements being one of test paths for the methods down the calling chain from the method category(). The argument OState is the state of the object right before the method invocation. The state consists of the list of attribute values of the object. The argument Arguments is the list of arguments to the method invocation. The argument Result is the returned value of the method invocation. The argument NState is the state of the object right after the method invocation.

The set of constraints on a test path includes the following four kinds of constraints: the class invariants on the attribute values *before* the method invocation, the preconditions on the attribute values and arguments *before* the method invocation, the postconditions on the attribute values and return values *after* the method invocation, and the class invariants on the attribute values *after* the method invocation. As an example, the set of constraints on the test path (the first test path enumerated) for "Equilateral" triangle objects consists of

```
invariants: {a@pre + b@pre > c@pre,
   a@pre + c@pre > b@pre,
   b@pre + c@pre > a@pre },
preconditions: { },
postconditions:
   {a@pre = b@pre, a@pre = c@pre,
   Result = "Equilateral"},
invariants: {a + b > c, a + c > b, b + c > a }.
```

The test data generator will convert the set of constraints on this test path into the following ECLiPSe clause:

```
triangleCategory([1|_], [OA, OB, OC], [ ], Result,
[NA, NB, NC]) :-
% Invariants
   OA + OB #> OC, OA + OC #> OB,
   OB + OC #> OA,
% Preconditions

% Postconditions
   OA #= OB, OA #= OC, Result = "Equilateral",
% State Updates
   NA #= OA, NB #= OB, NC #= OC,
% Invariants
   NA + NB #> NC, NA + NC #> NB,
   NB + NC #> NA.
```

If an attribute of the object is not specified in a postcondition, then it is assumed that the value of this attribute is not updated. Therefore, for each unspecified attribute a, there is an implicit postcondition a = a@pre for it. In the running example, there will be the following three constraints:

```
NA #= OA, NB #= OB, NC #= OC,
```

There will be five clauses for the predicate triangleCategory/5. Each of them corresponds to a distinct test path.

The predicate testClassMethod/5 is used to declare the domains for the attributes, arguments, and the return value, and initiates the constraint solving for a specific test path.

For the running example, the test data generator will generates a predicate testTriangleCategory(PathNo, OState, Arguments, Result, NState) as follows:

```
%Include constraint solving library
:- lib(ic).

testTriangleCategory(Path, OState, Arg, Result,
NState) :-
  Path = [N|_], N >= 0, N <= 5,
% Domains of variables
  [OA, OB, OC] :: -32768 .. 32767,
  [NA, NB, NC] :: -32768 .. 32767,
% Constraints on variables
  OState = [OA, OB, OC],
  NState = [NA, NB, NC],
  triangleCategory(Path,OState,Arg,Result,NState),
% Solving constraints
  indomain(OA), indomain(OB), indomain(OC),
  indomain(NA), indomain(NB), indomain(NC),
  locate([OA, OB, OC, NA, NB, NC], 0.1).
```

Query the ECLiPSe system using the following query:

```
  testTriangleCategory([1], OState, Arg, Result,
NState).
```

The ECLiPSe system will return

```
  OState = [1, 1, 1],
  Arg = [ ],
  Result = "Equilateral",
  NState = [1, 1, 1];
```

That is, the ECLiPSe system will return both the test input, [ ], and the expected output, "Equilateral" simultaneously. In addition, it will also return the state [1, 1, 1] of the object right before the method invocation, and the state [1, 1, 1] of the object right after the method invocation. Similarly, we can also generate the test input, expected output, and states for the remanding four test paths.

```
  OState = [2, 2, 1],
  Arg = [ ],
  Result = "Isosceles";
  NState = [2, 2, 1];

  OState = [2, 1, 2],
  Arg = [ ],
  Result = "Isosceles";
  NState = [2, 1, 2];

  OState = [1, 2, 2],
  Arg = [ ],
  Result = "Isosceles",
  NState = [1, 2, 2];

  OState = [2, 3, 4],
  Arg = [ ],
  Result = "Scalene";
  NState = [2, 3, 4];
```

If there is no solution for such a query, then the corresponding test path is an infeasible test path. The test data generator will generate the test input and expected output for a set of feasible test paths that satisfies the test coverage criterion. In the running example, all five test paths are feasible test paths and they together satisfy the all-decision test coverage criterion.

The above discussion covers the cases where a method does not call other methods. We now discuss the cases where a method does call other methods. If a method p with m test paths calls a method q with n test paths, there are two ways to test the method p. We can explore all feasible test paths in q for each feasible test path in p, namely, a total of $n \times m$ test paths. Or, we only explore one of the feasible test paths in q for each feasible test path in p, namely, a total of n test paths. In the first case, we can use the pair of predicates classMethod/5 and testClassMethod/5 for method p with a list of path identification numbers of length 2, e.g. [i, j], and for method q with a list of path identification numbers of length 1, i.e. [j]. This list represents the test path i in method p and the test path j in method q. In the second case, we are satisfied with a solution on any of the feasible test paths of method q. To handle this case, we use the path identification number 0 to represent any of the feasible test paths. This can be achieved by replicating all the clauses of predicate classMethod/5 with the path identification number to be 0.

For the running example, the test data generator will add five clauses to the predicate triangleCategory/5 with the path identification number argument to be [0|_]. The added clause corresponding to the first test path is as follows:

```
triangleCategory([0|_], [OA, OB, OC], [ ], Result,
[NA, NB, NC]) :-
% Invariants
  OA + OB #> OC, OA + OC #> OB,
  OB + OC #> OA,
% Preconditions

% Postconditions
  OA #= OB, OA #= OC, Result = “Equilateral”,
  NA #= OA, NB #= OB, NC #= OC,
% Invariants
  NA + NB #> NC, NA + NC #> NB,
  NB + NC #> NA.
```

In this case, we can use the pair of predicates classMethod/5 and testClassMethod/5 for method p with a list of path identification numbers of length 2, e.g. [i, 0], and for method q with a list of path identification numbers of length 1, i.e. [0]. This list represents the test path i in method p and any of the test paths in method q.

V. The 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 at a suitable state using the constructors of the class. It then generates code to invoke the method with the test input corresponding to this test path as arguments. Finally, it generates code to check if the actual output of the invoked method is the same as the expected output using the JUnit macro assertEquals.

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

```
public class TriangleTest extends TestCase
{
  public void testTriangleCategory()
  {
// path 1
  Triangle o1 = new Triangle(1,1,1);
  assertEquals("Equilateral", o1.category());

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

// path 3
  Triangle o3 = new Triangle(2,1,2);
  assertEquals("Isosceles", o3.category());

// path 4
  Triangle o4 = new Triangle(1,2,2);
  assertEquals("Isosceles", o4.category());

// path 5
  Triangle o5 = new Triangle(2,3,4);
  assertEquals("Scalene", o5.category());
  }
}
```

VI. Related Work

The Design by Contract is first proposed and designed in the programming language Eiffel [9]. The Eiffel contracts are part of the Eiffel programming language and can be executed directly. The AutoTest unit testing framework automates test case generation based on the Eiffel contracts [10]. The AutoTest framework uses the adaptive random testing to generate test input and executes the postcondition to generate expected output.

The Java Modeling Language (JML) is a Design by Contract specification language for Java [6]. The JML contracts are specified in Java syntax. The JML unit testing framework automates test case generation based on the JML contracts [4]. The JML framework uses the random testing and the constraint-solving on the precondition to generate test input and executes the postcondition to generate expected output.

The CASE tool AutoFocus is a model-based development tool for reactive systems [5]. The AutoFocus tool uses the System Structure Diagrams to specify the interaction structure of components and the State Transition Diagrams to specifies the behavior of a component. The AutoFocus tool uses the symbolic execution of constraint logic programming to generate test input and expected output [7, 13, 14]. Our approach is most similar to the approach of the AutoFocus tool.

VII. 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.

Acknowledgment

This work was supported in part by the National Science Council of R.O.C. under grant number NSC-97-2221-E-194-036.

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] Y. Cheon, A. Cortes, M. Ceberio, and G. T. Leavens, "Integrating Random Testing with Constraints for Improved Efficiency and Diversity," Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering, 2008.

[5] F. Huber, B. Schatz, G. Einert, "Consistent Graphical Specification of Distributed Systems," Proceedings of the Conference on Industrial Applications and Strengthened Foundations of Formal Methods, 1997, pp. 15-19.

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

[7] H. Lötzbeyer, A. Pretschner, and Er Pretschner, "AutoFocus on Constraint Logic Programming," Proceedings of (Constraint) Logic Programming and Software Engineering, 2000.

[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] B. Meyer, Eiffel: the Language, Prentice Hall, 1991.

[10] B. Meyer, H. Ciupa, A. Leitner, and L. Liu, "Automatic Testing of Object-Oriented Software," SOFSEM 2007: Theory and Practice of Computer Science, Springer, 2007, pp. 114-129.

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

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

[13] A. Pretschner and H. Lötzbeyer, "Model Based Testing with Constraint Logic Programming: First Results and Challenges," Proceedings of Second International Workshop on Automated Program Analysis, Testing and Verification, 2001, pp. 1-9.

[14] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel, "Model Based Testing for Real: The Inhouse Card case Study," International Journal on Software Tools for Technology Transfer, 5(2-3):140-157, March 2004.

[15] Standish Group, 2006 CHAOS Report, 2007.