ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2007, Vol.36, No.2

THE USE OF MODEL CONSTRAINTS AS IMPRECISE SOFTWARE TEST ORACLES

Šarūnas Packevičius, Andrej Ušaniov, Eduardas Bareiša

Software Engineering Department, Kaunas University of Technology
Studentų St. 50, LT–51368 Kaunas, Lithuania

Abstract. Many software test generation techniques target on generating software test data. Only a few of them provide automatic way to verify if software behaves correctly using generated test data. We propose a testing technique, which uses UML modeling language extension OCL as imprecise test oracle. Imprecise OCL constraints can be viewed as expressions which define expected results within some ranges of possible values. When software is executed using generated test data the output is verified against imprecise OCL constraints. If output invalidates imprecise OCL constraints, a tester can assume with some probability that software has bugs.

1. Introduction

High quality software is a desired goal for many software developers. But high quality software is an exception; defective software is norm [1]. The software testing is the most used way to ensure some level of software quality [1].

Software developers apply testing in development process to ensure an adequate level of quality for their products. In general, software testing consumes 50 or more percent of software project costs [2, 3]. Automation is the most popular way to reduce costs of software testing. Software developers usually try to automate in software testing:

- Tests execution.

  Tools automate execution of already prepared tests. Most popular tools are of record-playback type. They are used for automating user interface testing [2]. Another popular tools type is a unit testing framework. They are used for automatically executing created unit tests [4].
- Test data generation.

  Test data are generated using software code (white-box testing [2]) and functional specification (black-box testing [2]). There are many techniques which describe how to generate test data. Popular techniques are: random generation, path-based generation [5-11], execution based generation [12], generation using genetic methods [13-15], generation using formal and informal models [16-20].

Unfortunately many test generators generate only test data. They generate only test inputs, which has to be passed to software under test in order to test it. Many of these test data generation techniques are targeted to cover as much as possible code lines of software under test (code lines coverage) [2].

The full code coverage does not mean that software behaves correctly. After execution software with test data which covers all code we could only assume that a program works (does not crash). But we can not be sure that the program has produced a correct output. In this case the tester has to verify manually if the program has produced a correct output with given test data. The tester who decides if software under test has produced the correct output with given test data is called oracle [2, 3]. This work is manual and labour intensive.

2. OCL and tests generation

OCL standard stands for Object Constraint Language [21]. It was proposed by OMG organization. OCL allows UML [22] models to be extended with constraints. An UML diagram can not reflect all relevant aspects and constraints of a model. OCL standard has been developed to extend UML models by defining constraints.

For example, an UML class diagram can have an attribute of type integer. It is not possible to define limits of that attribute values or define relations with other attributes in the UML diagram. Let's suppose we want that this attribute could have its value within range of 0 and 100. Using OCL, this constraint could be implemented. Such constraints are called invariants. They have to be satisfied during all object lifetime.

OCL can also provide constraints for class methods, such as pre conditions for input values and

post conditions for output values. Let's consider a class Rectangle:

```
class Rectangle
{
        public int calcArea(int width, int height);
};
```

Let's define constraints for this method:

- Width has to be greater than zero
- Height has to be greater than zero

Let's define constraints for the result:

- Result is width multiplied by height

Constraints for a method calcArea such as width and height have to be greater than zero (pre condition) and the result is width multiplied by height (post condition) can be expressed in UML model by using OCL as follows:

```
context Rectangle::calcArea
 pre: width > 0
 pre: height > 0
 post: result = width * height
```

The post condition can perfectly serve as an oracle. The test generator can easily select two input values, which fall in the range of pre conditions, execute the unit under test and evaluate if software has calculated the result, which matches OCL post constraint. If the result matches, then the test has passed otherwise it has failed, thus revealing a bug in software.

Unfortunately, OCL constraints are not always available for all software units. Some constraints are not as precise as the one given in the example. In the given example, post condition completely reassembles the implementation of the method. When a post condition reassembles method implementation (provides full implementation) the constraint can be automatically transformed into software implementation [23]. In this case no testing could be needed. C. Philippe and R. Roger [24] proposed the idea how to convert OCL constraints directly to the code implementation in this way removing the need for testing of implementation, because implementation correctly represents the model. Constraints have to be precise in order to perform such automatic implementation generation. But it is not so common situation.

Aichenger and Salas [25] proposed a method for tests generation using OCL constraints. Their method relies on OCL specification mutation. In this method, inputs are selected that could detect changes in OCL specification. Such inputs passed to software under test could detect bugs in implementation too.

Korel and Al-Yami [26] presented a similar method, which injects constraints into software code. These constraints serve as an oracle inside code and allow verifying if some method implementation variables are within defined ranges.

Tracey et al. [27] presented a similar method for detecting bugs. They have analyzed "exceptions" feature of programming languages. Their method tried to calculate the required input, which would force to raise an exception, when the input is passed to software under test. If it is not possible to calculate such input, the software under test is bug free.

3. Test case generation using imprecise OCL constraints

OCL [21] constraints usually are imprecise, because it is easier to define imprecise constraints or even derive them form other model elements. We propose a test case generation technique using UML models and imprecise OCL constraints.

3.1. Imprecise constraint

Imprecise OCL constraint is a constraint, which defines the post or pre conditions not by exact formula, but by some boundaries or by a formula defining approximate result value. For example, in the previous example we could replace the post condition with the new imprecise one:

```
post: result > 0
```

This modification makes the OCL constraint imprecise. Despite the fact that such type of constraint is easier to define or derive, this constraint does not allow to verify if the computation was performed correctly. For example, we are testing an implementation of the method calcArea:

```
class Rectangle
{
        public int calcArea(int width, int height)
        {
      return width/height;
 };
};
```

There is usual mistyping in the given above implementation. The division '/' is used instead of multiplication '*'. Let's say data set (2, 3) was generated for input parameters width and height, respectively. The unit under test returns a value equal to 0.666. Using a precise constraint would reveal an error, but imprecise constraint does not.

We have another implementation with a bug, for example

```
class Rectangle
{
        public int calcArea(int width, int height)
        {
      return width-height;
        };
};
```

The error is misplaced subtraction operator in the given above code. Instead of the multiplication '*' symbol, subtraction '-'symbol was typed. When the test data (2, 3) are passed to the method, it returns -1. This value does not satisfy model constraint, and we have found the bug.

Imprecise OCL constraints can be deduced from other model elements. For example, we have the class Rectangle which represents geometrical shape and it extends the class Shape. Shape is a generic class and serves as an interface for all geometric figures. The Class Shape could define a method calcArea and have an OCL constraint associated with it. The OCL constraint defines that an area is always greater than zero or is equal zero. So this constraint has to be valid for all classes, which extend the Shape base class. The constraint for the Rectangle class is derived from the base Shape class in this example.

3.2. Testing procedure

During software testing using imprecise OCL constraints, we generate test data for each class method using some selected test data generation technique. Generated test data are filtered if they match input OCL constraints, thus reducing test execution time. The test data which satisfy OCL input constraints are passed to software under test and a software output is received. The received output is verified against OCL constraints. If the output does not satisfy OCL constraints, then we have detected a bug and the test has failed. If the output satisfies OCL constraints, then the test has passed.

The testing procedure is represented in the Figure 1. The procedure generates testing result as a set TR of records. Each record specifies the class C, the method M, the Boolean flag R (which defines does it contain bugs or not R), and the probability P of an inaccuracy of flag R.

```
Input. A set of classes, a set of methods in each
class, a set of OCL input constraints associated
with each method, a set of OCL output constraints
associated with each method.
Output. The set TR of records <C, M, R, P>
1. While there are untested classes
2.   Do select class C1
3.     While there are untested methods in class C1
4.       Do select method M1
5.         Select OCL constraint OCLI for
           method M1 input
6.         While can generate test data (the
           required coverage criteria has not
           been yet met)
7.           Generate test data TD
8.           If TD matches OCL input
             constraint OCLI then
9.             Execute M1 with test data TD
               and get result MER
10.             Select OCL constraint OCLR for
               method M1 output
11.            If MER does not satisfy OCLR
               then
12.              add <C1, M1, FAIL, 100%> to TR
13.              go to step 3
14.          Else
15.            Discard test data TD
16.        add <C1, M1, PASS,
           EVALUATEPROBABILITY()> to TR
```

Figure 1. Testing procedure

Test generation can be performed for selected method until the bug is found or until we have reached some defined coverage criteria. For example, if we selected all path criteria then testing continues by generating test input data until the input which forces method to produce invalid output is found (input does not satisfy OCL constraint) (lines 11, 12). Testing can also be finished when all paths in the method are executed and no inputs which have invalidated constraints have been found or no more new test data can be generated (line 14). If test passes, the EVALUATEPROBABILITY function is called which has to calculate the correctness of the testing result. If, using some test data, software under test produced an output which does not satisfy OCL constraints, then we definitely can say that the test has failed. If the output satisfies OCL constraints then we can only predict that test has passed. The EVALUATEPROBABILTY function calculates the probability that the test has passed. The probability depends on the number of test data executed. The probability is required because the OCL constraints are imprecise. Using imprecise constraints we cannot be sure completely if software calculated value is correct despite the fact that constraints are satisfied.

3.3. Testing framework

Software testing framework implementing our method is presented in Figure 2. Its main parts are: the test data generator, the oracle and models. The test data generator generates test inputs for software under test. The generated inputs are passed to software under test, which is executed using those inputs. Software under test produces some output.

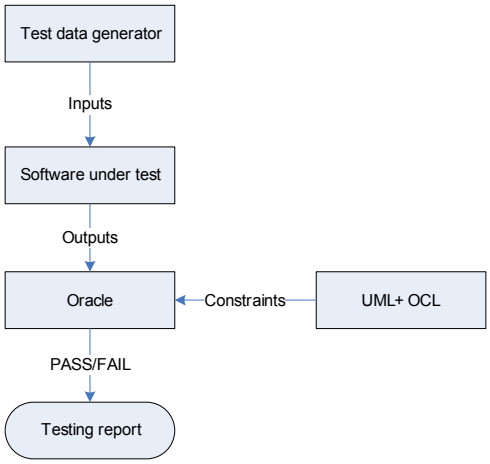

Figure 2. Testing framework

The oracle takes the produced output and evaluates its correctness against the UML+OCL model. It evaluates if the produced output satisfies provided OCL constraints and produces the testing report. If the output satisfies OCL constraints the PASS record is added to the testing report and the probability of the likelihood that test has passed is calculated. If the output does not satisfy the OCL constraint, the FAIL record is added to the testing report.

3.4. Test data generation

In order to generate test data for software under test any test data generation technique can be used. For test data generation random generation technique, genetic algorithms, symbolic execution [28, 29],

chaining approach [12], simulated annealing [30], variable dependence analysis [31], evolutionary algorithms [32] and other white box testing techniques could be used. The techniques could target to some code coverage criteria. For example, all paths criteria could be used as a mean for determining the test success probability.

OCL constraints can filter some generated test data and reduce testing time by doing that. For example, when we are generating test inputs for method calcArea, the random test generator chooses values from range -32K and +32K. Using OCL constrains values ranging from -32K to 0 can be removed because they do not satisfy pre conditions. By removing them the testing time is reduced.

3.5. Oracle

OCL post conditions serve as test oracles. Methods are executed using generated test data, and the output is checked against post conditions. If the function output does not satisfy post conditions, the test has failed and the bug is found. But if the result value satisfies the post condition, we can only predict with some likelihood that the test actually has passed and there are no errors in the implementation. The probability that the test has passed could be calculated using some of coverage techniques. For example, we have calculated that software under test has n paths in the code and using one of test data generation techniques, we have generated test data which covers m paths (0 < m <= n) of the code. After the execution we could get that all generated outputs satisfy OCL constraints and we have to calculate the success probability. Using all-paths coverage criteria, we can assume that p = m / n * a. Here p is the probability of accuracy of the testing result, m – the number of the covered paths with test data, n – the number of total paths in software under test, a – a constant (0 <= a <=1) which defines the imprecision level of OCL constraints. Here 1 could mean that OCL constrains are precise, and 0 could mean that we have no OCL constraints at all.

4. Example

We used our method for testing the Shapes example. Example software is represented as a UML class diagram in Figure 2. The base class Shape defines an interface for all child classes. For the simplicity and clarity of example, this class defines only one method for calculating the area size of the shape. There are two classes which implement this interface: Circle and Triangle. The Circle class has only one attribute r – the radius of the circle and one method SetR for setting the radius of the circle.

The class Triangle has three attributes: a, b and c. These attributes represent edge lengths of the triangle. The class under test has a method SetABC, which specifies edge lengths of the triangle. The Class Triangle extends the class Shape and implements its abstract method GetArea. The method GetArea calculates the area size of the shape.

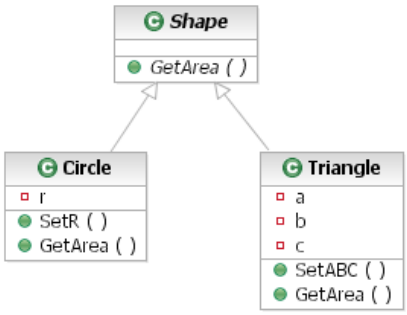

Figure 3. UML class diagram for the example system

The model has the following OCL constraints:

1. *context: Shape::GetArea*
2. *post: result >= 0*
3. *context Triangle::SetABC(a,b,c)*
4. *pre: a > 0*
5. *pre: b > 0*
6. *pre: c > 0*
7. *contect Circle::SetR(r)*
8. *pre: r > 0*

The constraint on the line 1 specifies that GetArea method have to return a result, which is a positive number. This constraint is automatically applied to classes Triangle and Circle through the class inheritance. The constraints on lines 4-6 and 8 define that input values have to be positive numbers.

The area size of triangle is calculated using the following formula:

$$S = b * c * \cos(\text{asin}(b*b + c * c - a * a) / (2 * b * c))$$

The area of circle is calculated using the following formula

$$S = Pi * r * r$$

Assume that the developer has provided an incorrect implementation of the Trinagle::GetArea method (asin function was replaced with acos function in this implementation):

```
class Triangle extends Shape
{
        public float GetArea()
        {
        return 0.5 * b * c * Math.cos(Math.acos( (b
* b + c * c - a * a) / (2*b*c) ) );
    };
};
```

The developer has provided a correct implementation for the Circle::GetArea method:

```
class Circle extends Shape
{
    public float GetArea()
    {
        Return Math.PI * r * r;
    };
};
```

The automatic test generator has generated some random inputs, software was executed using that input and the calculated software result was validated against OCL constraints. The testing results are given in Table 1.

Table 1. Testing results of the method Triange::GetSize

No.	a	b	c	Output	Result
1.	3	3	3	2.24	PASS
2.	4	3	3	0.5	PASS
3.	5	3	3	-1.75	FAIL

Using the last set of test data we have found an error. The implementation has calculated the -1.75 values, which does not satisfy the OCL constraint, and we can say that using the third set of input data, the test has failed. But the first and the second test data sets have provided the results which have obeyed OCL constraints and the test have passed. Therefore despite the fact that tests have passed, the calculated values are incorrect. In order to detect a fault in the implementation, the test generator has to generate a sufficient amount of test data. If the generator has managed to generate an input, which invalidates constraint, we have found a bug.

The random test data generator was also used for testing the Circle class. Class methods were executed using generated test data. The testing results are provided in Table 2.

Table 2. Testing results of the method Circle::GetSize

No.	r	Output	Result
1.	1	3.14	PASS
2.	2	12.56	PASS
3.	3	28.26	PASS

Using generated test data, we have not found any inputs which could force the implementation to produce an output which could invalidate OCL constraints. Thus we can assume that the implementation is bug free.

The testing report is provided in Table 3.

Table 3. Testing results of the example classes.

No.	Method	Result	Probability
1.	Triangle::GetArea	FAIL	100%
2.	Circle::GetArea	PASS	67%

The provided probability specifies that if we have failed test we are 100% sure that we have found the bug. When we have passed the test we can only say that the test result is accurate with some probability. The probability rises from imprecise model constraints. Because constraints define the method result imprecisely, the result of the passed test is imprecise as well. We can only evaluate approximately if the calculated value is really correct. The testing result is as precise as precise is the OCL constraint. The probability is less then or equal 100%. The probability depends on the number of test data executed. For example, if we decided to use all paths coverage criteria and have exercised all paths, the probability is quite high. If we have exercised only a few paths, the probability is low. All paths have been exercised during the testing of the Circle class, so the probability has to be high. If we assume that OCL constraint precision constant a is 0.67 the probability could be calculated as p = m/n * a * 100% = 1 / 1 * 0.67 * 100% = 67%. m and n equals 1 because the method has one path and the test generator has generated test data which allowed to cover all paths.

5. Experiments

In order to verify the efficiency of our testing method we have performed the testing on some benchmark programs. We have generated some mutants for each selected benchmark program. Mutations were created by substituting one operator with another in the code, changing constants values to other ones and by removing some statements from the code.

After performing mutations, we have performed the testing on mutated programs and have calculated what percentage of mutants our method can detect.

5.1. Benchmarks

For evaluating method besides the Triangle class, which was presented in Section 4, we have selected several popular benchmarks used by other authors [26]. The selected benchmarks and associated constrains are presented Table 4.

Table 4. Benchmarks for evaluation of the testing method

No.	Benchmark	Description	Constraint
1.	Triangle::GetArea	Calculates the area size of the shape	result > 0
2.	MinMax	Returns min and max values from the array of numbers	result.min <= any random input value & result.max >= any random input value
3.	Average	Returns the calculated average value of the array of numbers	result >= min & result <= max
4.	RestrictedAverage	Returns the calculated average value of the array of numbers excluding some values.	result >= min & result <= max
5.	Concat	Joins two strings a and b.	Result.length = a.length + b.length

5.2. Evaluation criteria

The percentage of killed mutants for each benchmark was selected as an evaluation criterion. The more mutants the method is able to kill the better it is. We also measured how many test data inputs we need to generate in order to kill mutants.

5.3. Results

During the evaluation of our technique we have used the random test data generation technique. Generated test data were submitted to mutated programs and the produced output was evaluated against OCL constraints. If the output produced by one mutant has not satisfied the OCL constraint we have marked that mutant as killed. Finally, we have calculated what percentage of mutants was killed for the selected program. The percentage of the killed mutants for each benchmark is provided Table 5.

Table 5. Killed mutants for the benchmark

No.	Benchmark	Mutants count	Killed
1.	Triangle::GetArea	9	89%
2.	MinMax	10	78%
3.	Average	9	100%
4.	RestrictedAverage	11	100%
5.	Concat	11	100%

During mutants generation some mutants appeared to be identical correct programs. Such mutants were removed from the set of mutants we were trying to kill. For example, such a mutant could be a program with removed variable initialization with zero value. In languages like Java each declared variable gets a default value of zero if it is of number type.

The percentage of the killed mutants shows that using imprecise OCL constraints is not enough to kill all mutants. But the percentage of the killed mutants is quite high. We are planning to extend the technique by adding one or two test cases with an actual input and the expected output values. This addition could increase testing accuracy and only slightly increases testing costs.

We have also measured how much test data are required to kill mutants for a specific benchmark. For example, during the Triangle class testing there were enough to generate only 10 inputs in order to kill mutants.

The number test data inputs used to kill Triangle class mutants is presented in Figure 3. The graph shows that a relatively small number of generated test data inputs is required for killing mutants and generating more additional data does not give better results. There is no difference if we have generated 1000 or 10 test data inputs the number of killed mutants remains the same for the Triangle class.

The same results could be observed and for Average program in Figure 4.

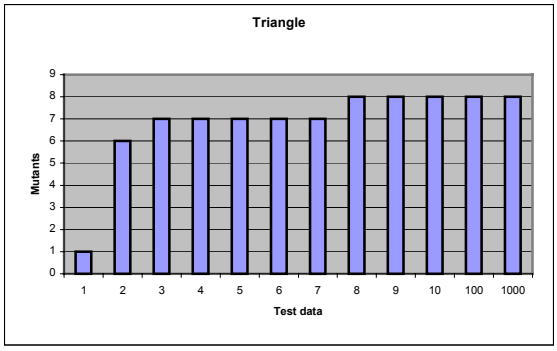

Figure 4. Test data input count needed for killing mutants of the class Triangle

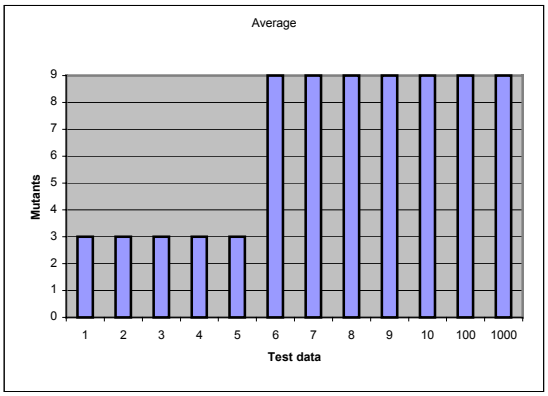

Figure 5. Test data input count needed for killing mutants of the class Average

6. Conclusions and future work

We have presented a method for performing software testing using imprecise OCL constraints. Imprecise OCL constraints are easier to define and derive from UML models. In order to detect an incorrect implementation we have to generate a sufficient amount of test data. By executing tests, we evaluate a software calculated output against OCL constraints. If constraints are not satisfied, we have established the fact of the existence of a bug in the implementation. The test data generation can be stopped at this point.

Out future work is targeted to developing a method for measuring the probability of passed tests more precisely. Also we are willing to determine what kinds of faults our method can detect. We are also thinking about extending this method in order to define new testing criteria. The criteria should define when to stop generating test cases and finish testing when we have not found any errors. We are interested in comparison of imprecise models with constraints versus models with precise OCL constraints in the testing aspect.

References

[1] **J. A. Whittaker, J.M. Voas.** 50 years of software: key principles for quality. *IT Professional*, *Vol*.4, 28-35, 2002.

[2] **B. Beizer.** Software testing techniques. 2*nd ed. New York: Van Nostrand Reinhold*, 1990.

[3] **R. V. Binder.** Testing Object-Oriented Systems: Models, Patterns, and Tools. *Boton: Addison Wesley Professional*, 2000.

[4] **E.N. Robert, H.P. Richard.** Unit testing frameworks. *Proceedings of the* 33*rd SIGCSE technical symposium on Computer science education*, *Cincinnati, Kentucky: ACM Press*, 2002.

[5] **G. Neelam, P.M. Aditya, S. Mary Lou.** Automated test data generation using an iterative relaxation method. *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, *Lake Buena Vista, Florida, United States: ACM Press*, 1998.

[6] **G. Arnaud, B. Bernard, R. Michel.** Automatic test data generation using constraint solving techniques. *Proceedings of the* 1998 *ACM SIGSOFT international symposium on Software testing and analysis*. *Clearwater Beach, Florida, United States: ACM Press*, 1998.

[7] **S. Nguyen Tran, D. Yves.** Consistency techniques for interprocedural test data generation. *Proceedings of the* 9*th European software engineering conference held jointly with* 11*th ACM SIGSOFT international symposium on Foundations of software engineering*. *Helsinki, Finland: ACM Press*, 2003.

[8] **B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, Q. Zhang.** Data Dependence Based Testability Transformation in Automated Test Generation. 16*th IEEE International Symposium on Software Reliability Engineering* (ISSRE'05) 2005.

[9] **W.E. Wong, L. Yu, M. Xiao.** Effective generation of test sequences for structural testing of concurrent programs. 10*th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS'05) 2005.

[10] **D. Beyer, A.J. Chlipala, R. Majumdar.** Generating tests from counterexamples. 26*th International Conference on Software Engineering* (ICSE'04) 2004.

[11] **A. Gotlieb, T. Denmat, B. Botella.** Goal-oriented test data generation for programs with pointer variables. 29*th Annual International Computer Software and Applications Conference* (COMPSAC'05), 2005.

[12] **F. Roger, K. Bogdan.** The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, *Vol*.5, 1996, 63-86.

[13] **F. Corno, E. Sanchez, M.S. Reorda, G. Squillero.** Automatic test program generation: a case study. *Design & Test of Computers, IEEE*, *Vol*.21, 2004, 102-109.

[14] **H. Harmanani, B. Karablieh.** A hybrid distributed test generation method using deterministic and genetic algorithms. *Fifth International Workshop on System-on-Chip for Real-Time Applications* (IWSOC'05) 2005.

[15] **P. Kalpana, K. Gunavathi.** A novel specification based test pattern generation using genetic algorithm and wavelets. 18*th International Conference on VLSI Design held jointly with* 4*th International Conference on Embedded Systems Design* (VLSID'05), 2005.

[16] **P. Amit.** Case studies on fault detection effectiveness of model based test generation techniques. *Proceedings of the first international workshop on Advances in model-based testing*. *St. Louis, Missouri: ACM Press*, 2005.

[17] **P. Alexander.** Model-based testing. *Proceedings of the* 27*th international conference on Software engineering*. *St. Louis, MO, USA: ACM Press*, 2005.

[18] **O.E. Mir, G. Hassan.** Model-based testing for applications derived from software product lines. *Proceedings of the first international workshop on Advances in model-based testing*. *St. Louis, Missouri: ACM Press*, 2005.

[19] **S.K. Kim, L. Wildman, R. Duke.** A UML approach to the generation of test sequences for Java-based concurrent systems. 2005 *Australian Software Engineering Conference* (ASWEC'05) 2005.

[20] **W. Xin, C. Zhi, L. Qi Shuhao.** An optimized method for automatic test oracle generation from real-time specification. 10*th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS'05), 2005.

[21] **J. Warmer, A. Kleppe.** The Object Constraint Language Second Edition: Getting your models ready for MDA. *Adisson-Wesley*, 2003.

[22] **M. Fowler, K. Scott.** UML Distilled. *Addison-Wesley*, 1997.

[23] **C. Philippe, R. Roger.** Towards Efficient Support for Executing the Object Constraint Language. *Proceedings of the Technology of Object-Oriented Languages and Systems*: *IEEE Computer Society*, 1999.

[24] **A.M. Brian, M.V. Jeffrey.** Programming with Assertions: A Prospectus. *IT Professional*, *Vol*.6, 2004, 53-59.

[25] **B.K. Aichernig, P.A.P. Salas.** Test case generation by OCL mutation and constraint solving. *Fifth International Conference on Quality Software,* 2005.

[26] **B. Korel, A.M. Al-Yami.** Assertion-oriented automated test data generation. *Proceedings of the* 18*th international conference on Software engineering*. *Berlin, Germany: IEEE Computer Society*, 1996.

[27] **N. Tracey, J. Clark, K. Mander, J. McDermid.** Automated test-data generation for exception conditions. *Software: Practice and Experience*, *Vol*.30, 2000, 61-79.

[28] **J. King.** A new approach to program testing. *International Conference on Reliable Software*, 1975.

[29] **J. King.** Symbolic execution and program testing. *Communications of ACM*, *Vol*.19, 1976, 385-394.

[30] **S. Kirkpatrick, C.D.G. Jr, M.P. Vecchi.** Optimization by Simulated Annealing. *Science*, *Vol*.220, 1983, 671-680.

[31] **M. Harman, C. Fox, R. Hierons, H. Lin, S. Danicic, J. Wegener.** VADA: a transformation-based system for variable dependence analysis. *Second IEEE International Workshop on Source Code Analysis and Manipulation*, 2002.

[32] **T. Back, F. Hoffmeister, H. Schwefel.** A Survey of Evolution Strategies. *Fourth International Conference on Genetic Algorithms, San Diego, CA*, 1991.

Received May 2007.