ISSN 1392 – 124X, ISSN 2335 – 884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2013, Vol. 42, No. 2

Test Data Generation for Complex Data Types Using Imprecise Model Constraints and Constraint Solving Techniques

Šarūnas Packevičius[1], Greta Krivickaitė[1], Evaldas Guogis[2], Dominykas Barisas[1], Robertas Jasaitis[1], Tomas Blažauskas[1]

[1] *Kaunas University of Technology, Department of Software Engineering, Studentų st. 50-406, LT-5136, Kaunas, Lithuania*

[2] *Singleton Labs, Studentų st. 65-307, LT-3000, Kaunas, Lithuania*

crossref http://dx.doi.org/10.5755/j01.itc.42.2.1855

Abstract. Number of software applications is growing rapidly, as well as their importance and complexity. The need of quality assurance of these applications is increasing. Testing is one of the key processes to ensure the quality of software and object-oriented applications in particular. In order to test large and complex systems, test automation methods are needed, which evaluate whether the software is working properly. The main goal is to improve effectiveness of object-oriented applications testing by creating an automated test data generation method for complex data structures.

This paper presents a test data generation method by adhering to software under test static model and its model constraints. The method provides an algorithm that allows generating test data for complex data structures, by analysing software under test model, its constraints and using constraint solving techniques for building corresponding test data objects and their hierarchies. The presented method is exemplified by simple case studies as well as a large I++ protocol implementing web service project.

Keywords: test data generation, complex data types, imprecise model constraints.

1. Introduction

In a typical software development organization, the cost of providing assurance that the program will perform satisfactorily within the expected deployment environments via appropriate debugging, testing, and verification activities can take from 50 to 75 percent of the total development cost [15]. Many of these activities can be automated, reducing their cost, increasing system quality, reducing the time to market and the maintenance costs.

The aim of this paper is to present an automated test data generation method for complex data structures. Object-oriented applications manipulate with various object hierarchies. Those applications in most cases operate with complex data structures, which in most cases are presented as objects aggregating or composing other objects. Traditional test data generation methods mainly generate tests for software unit under test (or parts of it) that use simple data structures, such as integers, floats, strings, arrays and pointers. However, these methods are unable generating meaningful or at least usable test data for unit testing of object oriented programs.

Most of real-world applications are composed of units working with complex data types as arguments. A test data generator generates input values for the selected method depending on the parameter type. For example, if the parameter is of type *signed short int* (in C++), the test generator will generate a value from the interval starting with –32,768 and ending with 32,767. If the parameter is of type *signed short int*, the generation algorithm is straightforward – the generated value has to be selected from the allowed range (the range depends on programming language and parameter type). However, there are more complicated types, and one of them is a *string*. The parameter of type string can be of any length. Test data generation algorithms, which are based on the values selection from allowed intervals, are suitable for generating values for types which are built-in into programming language. The usual types are: *integer*, *float*, *long*, *double*, *short*, *byte*, *char* and *string*. Real-world software implementations use complex types in most cases. The complex types are: arrays, lists, classes, interfaces, and structures.

The usual value generation algorithm for the parameter of type array or list is trivial. Generation for class or structure types is more complicated and requires a d ata flattening technique [25]. The parameter of a complex type is converted into a set of parameters which are of simple types.

The challenge comes when test data generation is applied for complex types and hierarchical class structures, as illustrated in the example in Figure 1.

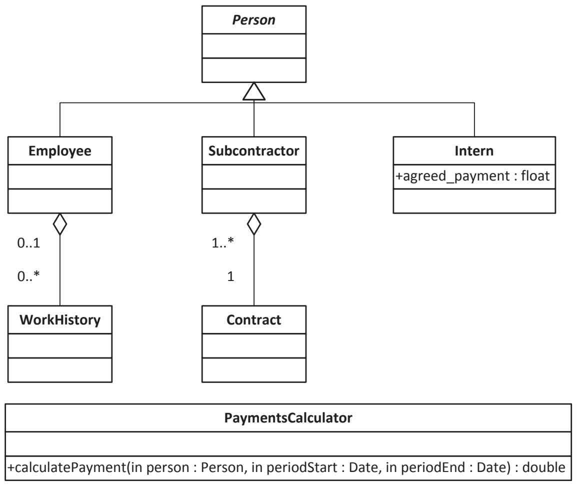


Figure 1. Class structure with various relations

The question is how to generate test data for method *calculatePayment*. Random values will not always fit since the implicit condition *periodStart* < *periodEnd* needs to be satisfied. Moreover, most of generators would use *null* value for *person* object as its type is an abstract class. Another case is the *Subcontractor* object, which wouldn't make sense without the relation to the *Contract* class.

This paper proposes solution to these problems by applying type flattening strategy, generating values for primitive types and filtering out those values, which do not match model constraints. Besides, it provides test data generation algorithm, presents software to be used for method evaluation, describes data generation end condition and the means for test data quantity reduction.

2. Problem statement

Figure 2 gives an example of the class Triangle, which is a part of the 3D renderer software. The class Triangle contains three attributes: a, b and c. Each attribute represents the triangle edge length and is of a float type. There also is the class Rasterizer with the method *render*. The method accepts two input parameters: the triangle and the texture and returns the array of Pixel objects. The triangle class is of a complex type.

The test generator cannot generate input values for the parameter triangle. To overcome this, the type flattening could be performed, method Render could be transformed into the one which accepts 4 parameters: *triangle_a*, *triangle_b*, *triangle_c* and *tex*. But this approach could have removed the meaning of a triangle.

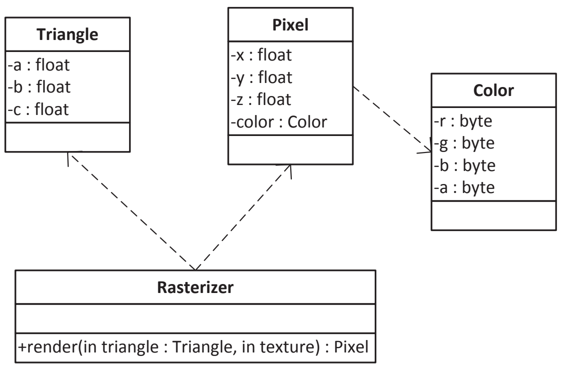


Figure 2. The 3d renderer software class diagram

The generator can generate invalid triangle objects. If the generator can understand the triangle object, it could adhere to some constraints, like: triangle edges have to be positive values; the sum of two edge lengths cannot be larger than the length of the remaining edge. In this case, the test generator just sees three float parameters and can generate any values, thus making some invalid triangles. This is similar to a possibility to generate an invalid float value and pass it as a p arameter value to a method under test. But this situation is not possible with simple types; the programming language syntax will not allow this to happen. But there is no means in programming language syntax to overcome a similar situation with complex types.

The main issues addressed in this paper are the following:

1. How to deal with complex data types and generate valid values at the same time. After the type flattening, the test generator could generate a l arger amount of invalid objects than an amount of valid ones. For example, if the random generator [7] is used for generating test data for testing Render method, it will generate a large amount of invalid triangles and just a few valid ones.
2. How to solve such cases, when the type of an input value is defined by interface or abstract class. In most cases, test data generators can provide empty (null) value [32].

3. Test data generation driven by static model constraints

If the test generator is aware of model constraints associated with complex types, it can generate only the valid test data. The test generator can differentiate between correct and invalid objects of complex types (for performing boundary value testing) and a required amount of valid objects of complex types. In this case the generator could create test data for parameters of complex types which is the same as test data for parameters of simple types.

For example, the class Triangle has the OCL [4] constraints presented in Figure 3.

```
1. context Triangle
2.   inv: a > 0
3.   inv: b > 0
4.   inv: c > 0
5.   inv: a + b > c
6.   inv: a + c > b
7.   inv: b + c > a
```

Figure 3. OCL constraints for the complex type Triangle in the Renderer software

Table 1. OCL constraints for the complex type Triangle fields in the Renderer software

No.	a	b	c	Valid	Constraint
1.	3	4	5	true	
2.	1	1	1	true	
3.	1	2	3	false	Invalidates the 5th OCL line.
4.	0	1	2	false	Invalidates the 2nd OCL line.
5.	34	30	3	false	Invalidates the 7th OCL line.
6.	10	10	14	true	
7.	12	12	20	true	
8	12	24	12	false	Invalidates the 6th OCL line.
9.	1	-1	1	false	Invalidates the 3rd OCL line.
10.	-1	0	-5	false	Invalidates the 2nd, 3rd and 4th OCL lines.

The OCL constraints for the Triangle class define that attributes *a*, *b* and *c* have to have positive values (lines 2-4). The OCL constraints also define what a correct rectangle is. The triangle edge length has to be smaller than the sum of the other remaining two edge lengths (lines 5-7). The generated values for Render method are presented in Table 1.

If the constraints are not used during test generation, all ten test data sets could be suitable for testing the Render method. But only the 4 test data sets are valid out of 10. In this case, too many values could only be used for testing the boundary conditions. The constraints for triangle class reduce an available set of input values: the constraint on the 2nd line reduces the set by half, the constraint on the 3rd line reduces the remaining set by other half, and the constraint on the 4th line reduces the remaining set by other half as well. All those eliminated halves could be replaced by only 1-2 boundary values. This could reduce test generation time, memory consumption and storage requirements for storing generated tests.

3.1. Generation algorithm

The proposed test data generation algorithm uses a complex type flattening strategy, generates values for simple types, assembles complex types from simple types and filters out unnecessary generated values, which do not match model constraints.

The test data generation algorithm for generating test data for single class method is presented in Figure 4.

For clarity, the test data generation algorithm is split into several parts. The test data algorithm takes a method description (extracted from a code or a model) as an input. The generator extracts all method parameters and their types and generates a test value for each parameter. After the test value is generated for a parameter, the generator checks if the value matches OCL pre-condition and if not, it is rejected and a new one is generated.

```
Input:  The Method M which accepts a set
        of parameters
        Pi (i=0;n).
        Parameter types Ti (i=0;n) for
        each parameter Pi.
        OCL constraints PC for the method M
        (pre-conditions)
Output: A set of generated values Vi (i=0;n)
        for each parameter Pi.

1. While there are method parameters left do
2.   Get  parameter type Ti
3.   value = determine the parameter type
4.        and generate the value for Pi, Ti.
5.   If (Vi does not match PC conditions)
6.   then
7.      Return to the step 3.
8.   else
9.      add value to the set V.
10.     Select next parameter Pi
11.     (go to the step 1.)
```

Figure 4. The test data generation algorithm for a class method

The process continues until a generated value matches OCL pre-conditions or the time limit for generating value is exceeded or generated invalid values count is exceeded, which is a safeguard for terminating infinitive generation. The generation is repeated for all method parameters. The generated test values for each parameter are assembled into a test data set, which in turn is stored for later usage or executed on software under test.

The test data generator uses several different test data generation algorithms. The algorithm has to be selected based on the parameter type. The algorithm selection is presented in Figure 5.

The test generator checks parameter type and selects the suitable generation algorithm. The algorithm is applied for such types as a simple type, a complex type and an array. When the parameter is of the pointer type in one case the generation algorithm for an array is used, in other case the generation algorithm for a parameter of the type, which the pointer refers to is used. Selected algorithm is different each time in order to generate data for both cases. If the element is of nested array type, the algorithm is applied recursively until other type is encountered.

```
Input:  The parameter P
        The parameter P type T
Output: The generated value V for the
        parameter P of the type T

 1. If T is a simple type then
 2.   V = generate simple value (T)
 3. If T is an array then
 4.   V = generate an array of values of
          the type T
 5. If T is complex type then
 6.   V = generate the value for the complex
          type T
 7. If P is pointer of type T then
 8.   If generation number is even then
 9.     V = generate the array of values
            of type T
10.   Else
11.     V = generate the value of type T and
12.         create pointer to it
```

Figure 5. Selection of test generation algorithm based on the parameter type

The test data generation for a parameter of a simple type is trivial and is presented in Figure 6.

```
Input:  The parameter type T
Output: The generated value V

1. Select the real number from the
   interval [0;1]
2. Scale the selected value to the interval
   allowed by the T range.
3. V = scaled value.
```

Figure 6. Test data generation for a parameter of a simple type

The test generator has to generate a value from the interval [0; 1]. The normal distribution generation can be used. Then the generator determines the allowed values range for parameter type and scales the generated value to match it. For example, if parameter is of the integer type, the allowed values range is [-32768; 32767]. For example, generator will select a value 0.1. The value 0.1 is scaled and it becomes -26214 (-26214 = (32767 – (-32768)) * 0.1 + -32768). If parameter is of the char type (Unicode, 16-bit), it is actually an integer value with-in a range [0; 65535]. The value generation is the same as for the integer type parameter.

The test data generation for a parameter of an array type is presented in Figure 7. The generator selects a length for an array. Then it executes the test data generation algorithm for each array value. The generated value is checked if it matches OCL constraints. If the generated value does not match, it is generated again. The value generation algorithm is executed until all array elements are generated.

The test data generation algorithm is used for generating values for parameters of a string type. Due to the nature of the string, which is just an abstraction of an array of character elements, the array generation algorithm is used. By execution the value generation algorithm for each array element, the generator can generate test data for parameters which are of the array type, where the array is composed of complex type elements.

```
Input:  The parameter type T
Output: The generated array V of elements
        of the type T

1. Select array length L
2. i = 0
3. While i < L do
4.   generate the value for the type T
5.   check if value satisfies OCL constraints
6.   If does not match then
7.      go to step 3.
8.   add generated value to the array V
9.   i = i + 1
```

Figure 7. Test data generation for a parameter of an array type

When the generation algorithms for simple types, array types, and pointer types are present, the test data generation algorithm can be defined for complex types. The test data generation for the parameter of the complex type is presented in Figure 8.

The test data generation algorithm for complex types reuses the algorithms for selecting generation algorithm, generating test data for simple, pointer and array type parameters. The test data generation algorithm for complex types takes as an input parameter type and OCL invariant constraints for that type. The OCL constraints are used to ensure that the correct object is constructed. The algorithm performs type flattening. At the first step, the empty object of parameter type is constructed. At this step the OCL constraints are not checked, because they obviously would be invalidated.

```
Input:  The parameter type T (class name)
        The list of the class T invariants
        INVi (i=0;n)
Output: The generated object V of the type T

 1. Select implementing type for the
    object V.
 2. Construct the empty class object V
 3. For each class field Fi (i=0;n) do
 4.   Get field type Ti
 5.   Generate the value for the field Fi
      of the type Ti
 6.   For each class invariant INVj
      (j=0;n) do
 7.      Check if the generated value
         satisfies INVj
 8.      If the value does not satisfy then
 9.         go to step 4.
10.   assign the generated value to
      the field Fi
11. For each class invariant INVj (j=0;n)
12. which defines relations between
    attributes do
13.    Check if the generated value
       satisfies INVj
14.      If value does not satisfy then
15.         go to step 2.
```

Figure 8. Test data generation for parameter of class/structure type

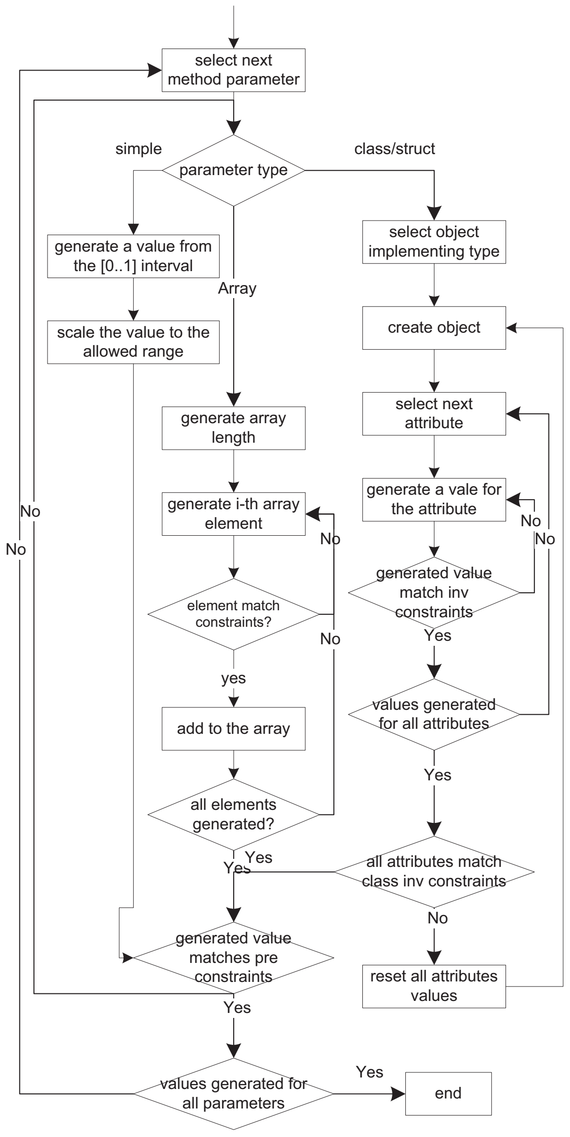
The generator extracts all attributes defined in the class and executes the test data generation algorithm selection and value generation algorithms for each attribute. If the attribute is of the complex type, the generator executes the same algorithm for value generation of the complex type, but in this case the other type is passed as an input to the algorithm. The value for the attribute is generated, and it is checked against OCL invariants. If the value satisfies all OCL invariants (defined for a class type and relevant to the attribute), the value is assigned to the object attribute. At this step, only invariants are checked which do not define relations between attributes (for example, inv: a < b + c is not checked at this step, only inv: a > 0 is checked). If value does not match OCL invariants, the new value is generated. The generation is repeated until the correct value is selected or time out is reached (to avoid deadlock conditions). This generation process is repeated for all object attributes. When all attributes are generated, OCL invariants are once again evaluated. In this case, the invariants which define relations between attributes are evaluated (these invariants cannot be checked in the first phase, because not all attributes have defined values). If the OCL invariants are not satisfied, the whole generated object is discarded and a new one is generated. The test data generation algorithm is provided in Figure 9.

3.2. Inheritance and interface handling

Test data generation for complex object types gets more complicated when the argument has a type of its parent. The object-oriented inheritance principle prevents generator from directly selecting required class and building its object. The software under test model could contain a set of classes that extend the argument class for which generator has to generate an object. If the type is a regular class (not an interface or an abstract class), the simplest case would be just to create object of required class and assign values for its fields.

More general solution is to find all classes in the model that directly or indirectly inherits class T (some parameter type). Then all classes that inherit type T are identified, the generator can choose any of them and instantiate the object. The next test case could take the same or the other class. The coverage criteria could also be extended by measuring what subset of possible classes was used for tests. In some cases, the list of classes, that inherit T class, could be empty. When such a situation is encountered, test generator can build a stub class adhering to model constraints. Constraints, that are valid for the base class, should be valid and for the stub class now.

The situation with interface could be handled in the same manner: finding implementing classes, selecting one of them or building a stub, as it is illustrated in Figure 10. It is possible to build the more intelligent stub by taking the model constraints into account.



select next method parameter
parameter type
simple
class/struct
Array
generate a value from the [0..1] interval
scale the value to the allowed range
select object implementing type
create object
select next attribute
generate a vale for the attribute
generated value match inv constraints
Yes
No
No
values generated for all attributes
Yes
all attributes match class inv constraints
No
reset all attributes values
generate array length
generate i-th array element
element match constraints?
No
No
yes
add to the array
all elements generated?
Yes
Yes
generated value matches pre constraints
Yes
No
No
values generated for all parameters
Yes
end

Figure 9. Test data generation algorithm for method parameters of complex types

The stub could randomly extend any of classes that extend class T or implements interface. The generator implements missing methods by providing empty method bodies. Then object is stubbed and generated, the OCL constraints are checked to verify if the object is correct. If constraints are not satisfied, the object and its stub are discarded, the algorithm is repeated again – a new abstract class or a new class is created as a stub with an implemented interface.

Inheritance and interface handling for test generation includes the following steps:

1. Find objects having a type of abstract class or interface;
2. Detect a set of extending or implementing classes;

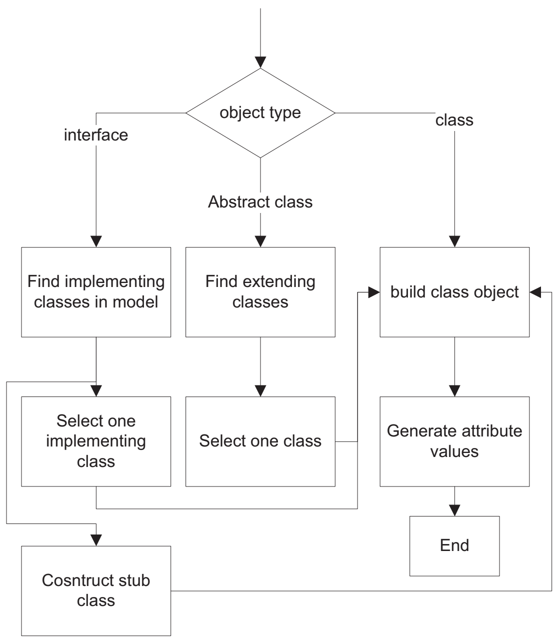


Figure 10. Object type selection and its value generation

3. Select a cl ass from inheritance hierarchy to use in object creation using random selection algorithm (with giving a higher priority for classes, which were not used in previous test data sets). This prioritized selection would allow to increase the generated tests code coverage;
4. Create complex type object and attribute values.

3.3. Test data generation end condition

The test generator has to overcome one critical problem – the object generation has to be completed in a finite amount of time. For deciding when to terminate test generation, test generation algorithm has to adhere to two problems:

- Generation of one test data set (object hierarchy for one parameter is fully generated);
- All test data sets generation.

Object hierarchies can contain circular references, for example, when object A points to B, and B points to A (Such a structure could be created adhering the composite design pattern [10]). The generator can endlessly generate object hierarchy in this case, as it follows the composition, and aggregation links in the model. To overcome this issue, the generator constructs the class composition graph and searches for loops in it. First of all, the loop is traversed only once, and for the next loop iteration the terminating values (null) are selected instead of building another set of objects.

In other cases, the object hierarchy could be too deep. To handle this situation, the generator can be parameterized with maximum depth value, thus limiting generation time. For example, generation depth of 1 means that values for a class with fields of simple type would be generated and fields of complex types would have assigned null values. The depth of 2 would mean that all values of class fields would be generated, and recursively algorithm would be invoked for all fields that are of complex types, but for those fields the objects would be constructed as for depth 1 – simple field would get values, complex ones would get terminating (null) value. The summarized hierarchy depth and loops handling in test data generation algorithm is presented in Figure 11.

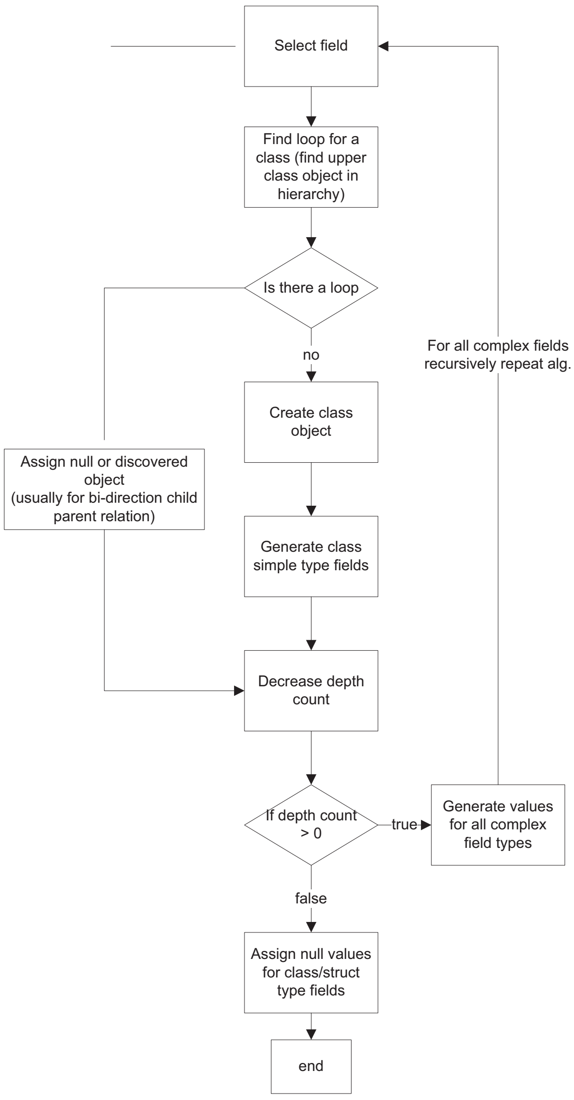

Select field
Find loop for a class (find upper class object in hierarchy)
Is there a loop
no
For all complex fields recursively repeat alg.
Create class object
Assign null or discovered object (usually for bi-direction child parent relation)
Generate class simple type fields
Decrease depth count
If depth count > 0
true
Generate values for all complex field types
false
Assign null values for class/struct type fields
end

Figure 11. Object hierarchy generation and loops handling

The whole test generation usually stops when the generator has generated a specific amount of test data. The tester specifies the needed amount of tests. The generator produces the required amount of tests and terminates. After the generation, the usual testing procedure occurs: the generated tests are executed, the code coverage is measured, and bugs are detected or not. When the tester has generated a certain amount of tests and bugs have not been found, the tester has to end testing assuming that the software under test is defect free or has to decide that another set of tests has to be generated. The tester has to make a choice: end testing or generate more tests and continue testing. The tester usually uses code coverage metric as a source for decision making on ending or continuing testing procedure. Based on the code coverage, the tests running time or the project schedule constraints, the tester decides when to end testing. The automatic test generator has no any influence on deciding when to end the tests generation.

The proposed test cases generation method uses the feedback driven test cases generation strategy. The test generator generates a t est data set, executes the software unit under test with the generated test data set, evaluates the testing result and testing metrics and decides if an additional set of test data is needed. The test generator uses the testing oracle (OCL constraints) as a means to evaluate if more tests are needed. The test generator has two different cases to evaluate:

- When the bug is found;
- When bug is not found.

When the bug is found, there is no point in generating more unit tests and executing them. The bug has been already found in the class method under test, and the test generator can work on testing other class methods and other classes.

When there are no bugs detected, the test generation could continue indefinitely. The proposed test generator uses the test pass accuracy coefficient and coverage as a stop condition. The test generation ends when one of the following conditions is met:

- The defined coverage is achieved;
- The coverage does not change after the subsequent tests generation.

For example, suppose that the test generator has generated 1000 test cases and executed them. After the execution, the code coverage is 78%, and the required coverage is 90%. Additional 1000 tests cases were generated and executed, but the coverage still remains at 78%. This situation could mean that there are unreachable branches in the software under test. If the coverage does not increase (after performing additional test cases generation and execution iterations), the testing process is stopped. These cases are reported as warnings for possible unreachable parts of code.

The test generator generates some test cases and executes them. If the bug has been found, the testing ends. If the bug has not been found, the code coverage is measured. If the selected coverage level has been achieved, the tests generation and execution ends. If the coverage has not been achieved, the time out value is checked. In this context, time out value means, if the code coverage level has not been changed after performing additional tests generation and execution steps, the testing has to be terminated.

The coverage criterion can be selected freely by tester and could be one of the following [3, 8]:

- All-branches;
- All-operators;
- All-code lines;

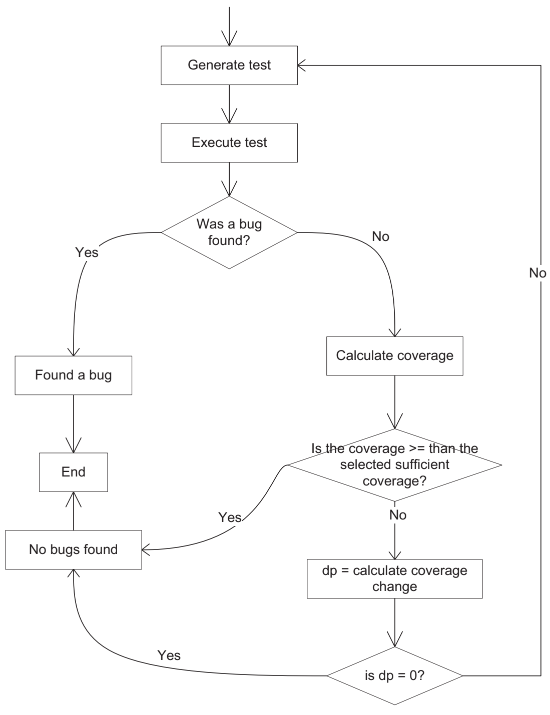


Figure 12. When to end tests generation

- The percentage of possible input values exercised;
- Model coverage.

When using OCL constraints for filtering input values, the amount of generated input values can be reduced. The tester can select a generation function for generating input values (for example, uniform distribution). Based on the generation function, the software under test could be tested with the most important values from the filtered interval. The test generation will continue until the selected percentage of possible input values are exercised. For example, if test input values are generated using a normal distribution function, all possible input values are generated with the same probability. If the tester chooses to generate values based on a normal distribution function, the majority of input values will be generated closely to average (most typical) value.

The tester can also decide when the testing should be stopped in case the coverage level is not increasing anymore. For example, the tester can decide that after 10000 additional test cases were generated and executed and the coverage has not increased, the testing should stop.

Besides the regular coverage criteria, the test generator method uses the model coverage criteria – the test generator tries to use as much as possible data model classes while generating test data. For example, if software model has 10 classes implementing one interface, the 100% coverage means that at least 10 test data sets were generated, and each of 10 classes were used in tests.

3.4. Constraint solving technique

While generating values for a cl ass fields, the backtracking [6] constraints solving algorithm is used. Test generator orders fields by their dependencies count. The generator selects a f ield with smallest dependencies value and generates initial value for it, finally, generator checks if the generated value satisfies model constraints. If the value does satisfy constraints, the value is accepted and the algorithm proceeds to the next field. If the value does not satisfy constraints, generator backtracks and selects new value repeating the process again. The backtracking steps are presented in Figure 9 (the backtracking steps are presented as negative decision transitions).

4. Test data quantity reduction

The input values count (that could be generated) depends on the programming language used for the implemented software under test. For example, if the class method takes only one parameter of an integer type, there are already 4294967295 possible input values. If the method accepts two parameters of integer type, the possible amount of input values is 4294967295^2. If the class method takes, for example, a parameter of complex type, the generated inputs count can be calculated by using the following formula:

$$fpv(type) = \begin{cases} c_{type}, when\_simple\_type \\ \prod_{i=1}^{n} fpv(attr_i), otherwise \end{cases} \qquad (1)$$

here, *fpv(type)* – the amount of possible input values for the type "*type*"; c_{type} – the amount of possible input values for the simple type "*type*", selected from Table 2; $attr_i$ – the type of the attribute which is a part of the "*type*" type.

The formula flattens the complex type and multiplies all the input values counts, which could be generated for each complex type attribute. For example, if there is the complex type Vector4D, which is composed of 4 attributes: x, y, z, and w (each attribute is of a float type), the input values count is calculated as follows:

$$fpv(Vector4D) = (fpv(float))^4 = 3{,}4 \cdot 10^{34} \qquad (2)$$

Table 2. The input values counts for some types

No	Type	Input values count
1.	Integer	2^{32} = 4294967296
2.	Float	2^{32} = 4294967296
3.	Double	2^{64} = 18446744073709551616
4.	Long	2^{64} = 18446744073709551616
5.	Char	2^{8} = 256
6.	Byte	2^{8} = 256
7.	Vector3d	2^{32*3}= 79228162514264337593543950336
8.	Triangle	2^{32*3}= 79228162514264337593543950336

Thus, even for a method which accepts only one input parameter of quite not very complex type, the generated input values count is quite big. The amount of possible input values for a method which accepts n input parameters can be calculated using the following formula:

$$Tpp = \prod_{i=1}^{n} fpv(p_i) \qquad (3)$$

here, *Tpp* – the total generated input values count for a method of *n* input parameters; p_i – the *i*-th input parameter type; *n* – the number of input parameters for the method.

Using OCL constraints, the generated input values count could be reduced. The reduction level depends on the OCL constraints. The generated input values count can be calculated using the following formula:

$$Tpp' = \prod_{i}^{n} fpv(p_i) \cdot cr_i \qquad (4)$$

here, *Tpp'* – the total generated input values count for a method of *n* input parameters; p_i – the *i*-th input parameter type; *n* – the number of input parameters for the method; cr_i – the reduction level of input values for the *i*-th parameter.

The reduction level depends on the OCL constraints, associated with the method parameter or parameter type, precision level. If the constraints are precise, only a few values are used (boundary values). If the constraints are not available – there is no reduction at all. The reduction level can be within bounds:

$$0 < cr_i <= 1. \qquad (5)$$

The cr_i value of 1 means that there is no reduction at all (thus there is no OCL constraints constraining a method input parameter or its type). The value 0 could mean that all possible input values are eliminated, but this case is not possible (because at least one boundary value has to be used). The cr_i value can be calculated using the 7-th formula. This formula depends on the a_i value, which is calculated as follows:

$$a_i = \sum_{j-1}^{n} \frac{0.5}{2^{j-1}}, \qquad (6)$$

here, a_i the precision level of OCL constraints associated with a class field or method, result value; *n*– the count of ">", "<", ">=", "<=" operators.

$$cr_i = \begin{cases} 1, a_i \geq 1 - cb \\ 1 - a_i + cb, otherwise \end{cases} \qquad (7)$$

here, cr_i – the input values reduction level; a_i – the precision level calculated using the 6-th formula; *cb* – the percentage of input values allocated for testing boundary conditions (it is usually 0,01, and cb has to be greater than 0).

The generated input values count can be reduced by several times. The reduction level can be calculated by dividing the amount of generated input values count without using OCL constraints by the generated input values count using the OCL constraints as follows:

$$r = \frac{Tpp}{Tpp'} = \frac{\prod_{i=1}^{n} fpv(p_i)}{\prod_{i}^{n} fpv(p_i) \cdot cr_i} = \frac{1}{\prod_{i=1}^{n} cr_i} \tag{8}$$

The reduction level depends only on the available OCL constraints and their precision levels. If there are no OCL constraints, the cr_i values are equal to 1, and r value becomes equal to 1. The generated input values count is not reduced in this case.

The reduction level can be calculated for the example system presented in Figure 2, which also has OCL constraints presented in Figure 3as follows:

$$Tpp = 2^{32} \cdot 2^{32} \cdot 2^{32} = 7{,}9 \cdot 10^{28} \tag{10}$$

$$cr_1 = cr_2 = cr_3 = 1 - 0{,}75 + 0{,}01 = 0{,}26 \tag{11}$$

$$Tpp' = 2^{32} \cdot 0.26 \cdot 2^{32} \cdot 0.26 \cdot 2^{32} \cdot 0.26 \approx 1{,}3 \cdot 10^{27} \tag{12}$$

$$r = \frac{1}{0.26^3} \approx 56.8 \tag{13}$$

The cb value was equal to 0.01 when the 7th formula was applied. The one percent of possible input values is allocated for boundary conditions testing. The OCL constraints precision level a_i was calculated using the 6th formula and is equal to 0.75. The reduction values are the same for all method parameters; their values are equal to 0.26. After the calculation with all values included, the reduction is 56.8. This means that by adding 6 OCL constraints for the class Triangle, the generated input values count is reduced by 56.8 times.

5. Metrics

This paper presents the theoretical model of test data generation using model constraints. However, the proposed test data generation method does not depend on any specific coverage criteria. The coverage criteria could be any of those:

1. Code coverage;
2. Path coverage;
3. Model coverage and evaluation of the amount of existing classes and interfaces, which were used from the model.

The following section provides the method evaluation including an example model and OCL constraints.

6. Evaluation

The test generation method was assessed at testing a commercial application. The application under test was implemented using Java programming language as an xml web service. The application was an implementation of a quality parameters exchange protocol. The quality exchange protocol (I++) was developed by majority of European automakers. The protocol (and an application itself) is designed to allow transferring selected automobile quality parameters from CAD (Computer Aided Design) application to factory production software for measuring the selected quality parameters. The application is providing a simple interface: just a set of several methods in one class for loading, transforming and storing quality parameters data. The complexity lays in the method parameters; each function takes as an input and returns a result of a complex data structure. The data structure is defined in 129 different classes and its instance is composed of several hundred objects that are interconnected between each other. The fragment of I++ DMS [40] data model is presented in the following figure.

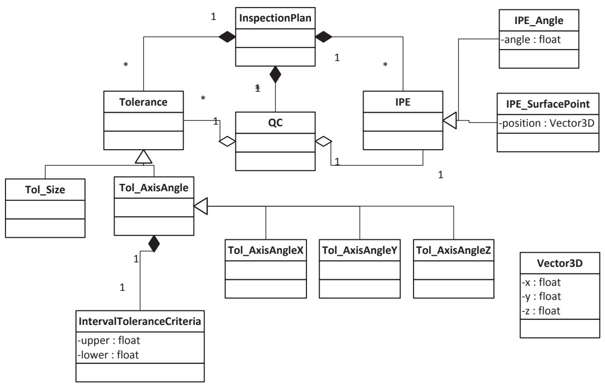


Figure 13. The fragment of I++ DMS data model

The subset of measurement plan is presented in the figure, which models the whole measurement plan (InspectionPlan class) for one part of the car, and defines a s et of possible features that could be measured on the car body. The possible features have types (in this fragment the angle and surface point features are visible). The angle feature is presented as the IPE_Angle class and uses a co mposite design pattern [10]. It aggregates any other two features (specifying the angle between the selected two features). The surface point feature (defined by IPE_SurfacePoint) only contains the coordinates in 3D space. Each feature contains nominal values (defined by IPE classes and their attributes) and allowed manufacturing deviations, tolerances that specify in what range the manufactured part features could fall. For example, a surface point feature position can vary in some interval by x, y, and z axes; an angle value can vary in three intervals parallel to x, y, and z axes. The nominal and tolerance values are joined by QC object (quality control) that connects IPE extending objects and relevant Tolerance extending objects. The InspectionPlan object in turn aggregates all the QC objects, thus containing the full measurement plan for one part of the car. The static model by itself does not contain information what objects could be aggregated by QC object. For example, if QC aggregates IPE_SurfacePoint object, it i s not allowed for it to aggregate Tol_AxisAngle or its child classes, as that would not make sense in measurement plan. To overcome this issue, the model is enhanced by using OCL model constraints. For this small model fragment, the OCL constraints are presented in the following figure.

```
1.  context InspectionPlan
2.   inv invariant_InspectionPlan1:
3.    qc->forAll(qc : QC |
      qc.ipe.oclIsTypeOf(IPE_Angle) = true
      implies qc.tolerances.length = 3 and
qc.tolerances[0].oclIsTypeOf(Tol_AxisAngleX)
= true and
qc.tolerances[1].oclIsTypeOf(Tol_AxisAngleY)
= true and
qc.tolerances[2].oclIsTypeOf(Tol_AxisAngleZ)
= true)
4.   inv invariant_InspectionPlan2:
5.    qc->select(qc : QC |
      qc.ipe.oclIsTypeOf(IPE_Angle) = false)
6.   inv invariant_InspectionPlan3:
7.    qc->forAll(qc : QC |
qc.ipe.oclIsTypeOf(IPE_SurfacePoint) = true
implies qc.tolerances.length = 3 and
qc.tolerances[0].oclIsTypeOf(Tol_AxisPosX) =
true and
qc.tolerances[1].oclIsTypeOf(Tol_AxisPosY) =
true and
qc.tolerances[2].oclIsTypeOf(Tol_AxisPosZ) =
true)
8.
9.  context QC
10.  inv invariant_QC1:
11.   ipe->forAll(ipe: IPE |
      parent.ipe.contains(ipe))
12.  inv invariant_QCCheckTolerances:
13.   tolerances->forAll(tolerances:
      Tolerance |
      parent.tolerances.contains(tolerances))
14.
15. context TolAngle
16.  inv invariant_TolAngle1:
17.   value.upper > value.lower
18.  inv invariant_TolAngle2:
19.   value.upper <= parent.ipe.angle
20.  inv invariant_TolAngle3:
21.   value.lower >= parent.ipe.angle
22.
23. context IPE
24.  inv invariant_IPE:
25.   name.regExpMatch('[1-9][0-9]{11}')
```

Figure 14. I++ DMS data models OCL constraints

The model defines that InspectionPlan object composes all the IPE, Tolerance, and QC objects, and each QC object only aggregates Tolerance and IPE objects from the InspectionPlan lists. This constraint is presented in lines 9-13. Those constraints state that every object referenced by QC object, must be contained in InspectionPlan lists. The other complex relation is modeled in the 3rd line. That constraint states that if the QC aggregates IPE_Angle object, it also is required to aggregate 3 Tolerance objects: Tol_AxisAngleX, Tol_AxisAngleY, and Tol_AxisAngleZ. As well, the IPE_angle is not allowed to aggregate itself (line 5), it cannot measure angle between itself and some other features, and as well it has to reference two different features. The similar situation is for the IPE_SurfacePoint object: if QC aggregates this object, it also have to aggregate Tol_AxisPosX, Tol_AxisPosY, and Tol_AxisPosZ (not visible on a diagram snippet) objects (line 7). Another constraint defines that for angle features the nominal value has to be within tolerance bounds (lines 15-21). The final constraint (lines 23 - 25) defines that features name (IPE.name) has to match certain regular expression – name is four numbers string that does not start with 0.

The test data were generated for one single method "storeProductStructure" in the class IppWebService. This method allows exchanging quality data between the car manufactures and their suppliers. This method accepts just several parameters: inspectionPlan, and sender.

The sender parameter is a simple one: it consists of the application name and the user name who are calling web service method. The inspection plan is more complex. It is the root object that aggregates all other objects. It represents measurement plan for one car part. The related model constraints for I++ DMS model are presented as OCL constraints. OCL constraints for all objects are written manually using Eclipse with OCL plugin and placed in a s eparate single file.

The test generator has analysed the model and constructed several test data sets, presented in Figure 15 as an object diagram.

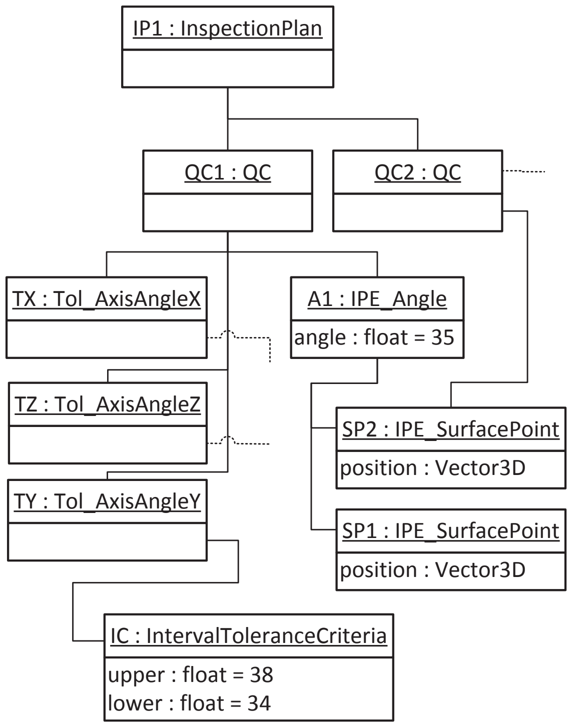


Figure 15. Test data set as generated inspection plan object structure

The generator starts from the root object and constructs InspectionPlan object. This object is supposed to compose several QC objects. To satisfy this requirement, generator randomly selects QC list size (2), and generates QC objects in the loop. The first QC object is supposed to aggregate one IPE object. To satisfy this requirement, test data generator randomly picks one of the classes that extend IPE class (in this case it is IPE_Angle).

Then test generator repeats test data generation algorithm and builds IPE_Angle. While generating the IPE_Angle object, an arbitrary value is selected for the angle field. Also two additional objects have to be created that. They are aggregated by IPE_Angle object. Those two object types are randomly selected from classes that extend the IPE class, and the test data generation algorithm is performed for them again. As the last IPE_SurfacePoint objects are the terminating nodes, the test data algorithm returns back and generates the remaining Tolerance objects that are required by QC object.

As the QC now aggregates the IPE_Angle object, the OCL constraints allow only generating 3 tolerance objects of Tol_AxisAngleX, Tol_AxisAngleY, and Tol_AxisAngleZ types, thus again repeating test data generation algorithm for those objects. As each Tol_AxisX,Y,Z object aggregates InternalToleranceCriteria object, the generator has to generate those objects as well. As generator generates InternalToleranceCriteria objects, it b uilds random object and checks if upper and lower values match constraints (navigating through model till the IPE_Angle object and checking if upper and lower values match the IPE_Angle.angle field value). If upper and lower values are incorrect, the generator discards them and selects new ones. As IntervalToleranceCriteria objects are constructed, the generator retreats back for generating inspectionPlan objects and generates remaining QC objects using the same approach, but in this case it can pick other random IPE_Types, for example, it c an pick IPE_SurfacePoint type and in result it will build a different objects sub-hierarchy. Then InspectionPlan object is generated, the generated data structure creation steps are printed as JUnit [23] statements, and calls for the storeInspectionPlan method are generated. The test data generator can mark the generated object hierarchies as the ones that are valid (matches all model constraints) and the ones that are invalid (do not match constraints). It is expected for unit test to fail using invalid data, and to succeed using valid ones. Thus it serves as an initial test oracle.

7. Related work

7.1. Random test data generation

The simplest test data generation strategy is random test data generation. Test data are created by selecting input values randomly [7] for software under test methods and checking if generator has reached the defined coverage criterion. In this case, test data contain mostly meaningless data (from software domain perspective).

The authors proposed modifications to random generation technique which performs feedback analysis of tests execution results [29, 32].Tests are executed and the coverage is calculated. Based on the coverage level, the decision is made if next random values have to be generated.

The advantage of this approach is that the generation algorithm is quite simple and easy to implement. The drawback of this approach is that the generation is a t ime consuming process, especially when the software unit under test is quite complex.

7.2. Path based tests generation

Software control flow charts are created during the unit analysis. Test inputs are generated using graph theory methods. Generated inputs drive the software execution by some paths. The input data selection algorithm is the main part of the generator. The selected data would force the code to be executed into the selected code branch. For data selection the constraints solving techniques [12] and the relaxation methods are used [14]. These methods select initial values and, based on the feedback from software execution metrics, perform selected values adjusting. Unfortunately, these approaches work only with values inside the unit under test which are of a simple type (float, integer, etc..) and are not capable to handle units which are calling other methods, functions and/or operate with variables of complex types (arrays, pointers, data structures, etc..). Gotlieb et al. have proposed method for test generator, which can generate tests for software which uses pointers [13]. There are similar methods for testing software which calls procedures and/or functions [21, 35]. It is proposed to use data transformation into equivalent data types, while testing software, which operates with complex data types, and then use existing generation methods [22, 26]. The advantage of path-based test generation is the possibility to generate the minimal needed test data set which would satisfy the selected coverage criterion. Also, during code analysis, the unreachable paths of code could be detected and marked as failures [2]. The disadvantage of path-based test generation algorithms is that they are quite complex and not always guarantee a full code coverage.

7.3. Dynamic testing

The test generation is based on the data gathered during software unit execution instead of data collected during static code analysis. The software unit is executed with some input data and during its execution the runtime parameters are observed: executed paths, executed branches, executed operators. Based on observations, the new additional input data are generated in order to drive execution by selected control flow path [8]. F erguson et al. are proposing various methods for improving code coverage by tests, such as the chaining approach [8]. Hierons et al. are proposing the program slicing by diving software unit into separate branches [18]. The main drawback of these approaches is that the execution of software has to be performed, which requires the preparation of the whole software infrastructure (environment). This preparation cannot be performed automatically. The advantage of those methods is that tests can be generated very quickly.

7.4. Search-based software testing

The genetic algorithms can be adapted for test generation [5, 16, 19, 33]. The initial set of tests cases is created, after that tests are executed and their efficiency is measured using the selected coverage criterion. During the next iteration, child tests are generated by selecting better performing tests and killing less successful tests. Using this approach, tests are created which achieve the selected coverage criterion with less test data or with less testing time. The advantage of these approaches is that test generation is fast, the main drawback: there is no guarantee that the defined goal of test generation will be reached at all.

7.5. Model-based generation

Tests can be generated when the implementation of software under test is still not present. Tests are generated using software models. Formal and informal models can be used as a source for test generation. It is also possible to generate tests directly from requirements specification. But in this case the models are needed anyways. These models could be transformed from requirements specification, even from the textual ones [11]. During the test generation using formal specifications, the black box methods can be used, such as boundary values analysis and average values analysis. Model-based test generation is increasingly becoming more and more important due to the emergence of the model driven engineering [36] and the model driven development [24] methods.

Formal specifications, expressed in the Z notation [34] and others [30], strictly define the software functionality. Using a formal software specification, it is possible to generate tests for that software. The formal specifications allow generating not only test data but can also provide an oracle which would be able to determine if software works correctly with given test data. Due to the fact that formal specifications are used for defining critical systems and real time systems, their testing can be alleviated by generated tests from formal specifications [39]. The disadvantage of those methods is that creating formal specifications is expensive and only a few projects were developed using such strategy.

The Unified Modeling Language (UML) [9] is semi-formal modeling language. These informal models have some features which could be handy during test generation. These models are called tests-ready models [27]. They are usually extended to some extent in order to be suitable for test generation. For example, UML has a t esting profile [1]. An Object Constraint Language (OCL) [4, 37] model (in addition to UML models) could be used for test generation. Informal models are actively used for testing software developed using product lines approach [27].

Tests generated using software models usually try to examine such cases as: missing action, incorrect data manipulation by overrunning buffers, incorrect data manipulation between class boundaries, incorrect code logic, incorrect timing and synchronization, and program code statements execution it incorrect sequence.

It is possible to transform software models into graphs, such as state graphs. For example, UML diagrams (state or sequence diagrams) can be used for test generation by transforming them into graphs. When the graphs are created, the usual test generation techniques (for testing program code) can be used [31]. Kim et al. are also proposing to transform models from one language into other ones. Target languages are more suitable for test generation, for example, the UML models are transformed into SAL (Symbolic Analysis Laboratory) models and SAL models are used to generate tests [20].

Oriat has proposed the Jartege tool and a method for random generation of unit tests for Java classes defined in JML (Java Modeling Language) [28]. JML allows writing invariants for Java classes and pre and post-conditions for operations. JML specifications are used as a t est oracle and for the elimination of irrelevant test cases. Test cases are generated randomly. Proposed method defines how to construct test data using class constructors and methods calls for setting the object initial state.

7.6. Combined techniques

It is possible to mix code based and model based test generation methods together. It is not always possible to have the full specification of software under test. In order to test this software the mix of code-based test generation and model-based test generation methods can be used. Visser et al. have proposed a path finding tool [38]. Data structures are generated from a description of method preconditions. The generalized symbolic execution is applied to the code of the precondition. Test inputs are found by solving the constraints in the path condition. This method gives full coverage of the input structures in the preconditions. Then the software code is available, the code is executed symbolically. A number of paths are extracted from the method. An input structure and a path condition define a set of constraints that the input values should satisfy in order to execute the path. Infeasible structures are eliminated during input generation.

There are also methods for combining both techniques together [2]. Test data are generated based on code-based generation techniques. Software is executed with generated test data, then it is checked if software has entered the undefined in the model state, or has exceeded restrictions that are defined for its variables [17]. Based on code and specification, it is possible to verify if code paths executed during testing are defined in the model and allow to check if software has not changed its state to the undefined in the model or has performed illegal transition from one state to another one, thus violating specification [2].

8. Conclusions and future work

The OCL constraints can be used for filtering test data. Using the OCL constraints and the proposed generation method, a l arge amount of meaningless values is filtered out from the generated test data. The more meaningless test data are removed, the faster tests can be executed. The invariant and pre-condition OCL constraints can be used for filtering the generated test data. The pre-condition OCL constraints filter which test data could be passed to the software under test. The invariant constraints can be used to ensure that only the valid objects of complex types could be passed to the software under test.

Using the defined algorithm, incorrectly constructed objects of complex types could be detected early and not used for testing. The constructed complex object is checked against invariant constraints. If the object is not valid one, the test generator could discard, or use as the boundary value in tests. By knowing which object is valid and which one is not, the generator can limit the amount of test data, which use unnecessary boundary values. Even for a trivial sample, the test data quantity could be reduced by 50 times;

The use of a f eedback driven test generation technique prevents from test never ending test generation and execution situation. When the bug is detected, the test generation and execution finishes. When bugs are not found, the test generation and execution continues until the chosen coverage a criterion is met. After generation and execution of some tests, the coverage change is measured. If the coverage is not changing, the testing ends. This happens when the code contains unreachable branches;

We have presented the theoretical model of test data generation using model constraints. The future work includes improving and evaluating test data generation method by using other constraints solving algorithms instead of simple backtracking one. The future works as well includes evaluating method by testing full I++ DMS service and measuring selected coverage.

Acknowledgments

The work described in this paper has been carried out within the framework the Eureka ITEA2 project ATAC: Advanced Test Automation for Complex Software-Intensive Systems (2011- 2014) , funded by the project "The 'Eureka' research and development projects - Eureka" VP1-3.1-ŠMM-06-V-01-003 of the European Social Fund (ESF) H uman Resource Development Programme of Lithuania.

References

[1] **P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, C. Williams.** Model-Driven Testing: Using the UML Testing Profile. *Springer-Verlag Berlin and Heidelberg GmbH & Co. K,* 2007.

[2] **D. Beyer, A. J. Chlipala, R. Majumdar.** Generating tests from counterexamples. In: *26th International Conference on Software Engineering (ICSE'04)* 2004. pp. 326-335.

[3] **J. J. Chilenski, S.P. Miller.** Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 1994, Vol. 9, No. 5, pp. 193-200.

[4] **T. Clark, J. Warmer.** Object Modeling with the OCL: The Rationale behind the Object Constraint Language (Lecture Notes in Computer Science). *Springer,* 2002.

[5] **F. Corno, E. Sanchez, M. S. Reorda, G. Squillero.** Automatic test program generation: a cas e study. *Design & Test of Computers, IEEE*, 2004, Vol. 21, No. 2, pp. 102-109.

[6] **R. Dechter, D. Frost.** Backtracking algorithms for constraint satisfaction problems, 1999, University of California at Irvine: Technical Report.

[7] **J. W. Duran, S. C. Ntafos.** An evaluation of random testing. *IEEE transactions on software engineering*, 1984. 10(4): 438-444.

[8] **R. Ferguson, B. Korel.** The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 1996, Vol. 5, No. 1, pp. 63-86.

[9] **M. Fowler.** UML Distilled: A Brief Guide to the Standard Object Modeling Language. *Addison-Wesley Professional,* 2003.

[10] **E. Gamma, R. Helm, R. Johnson, J. Vlissides.** Design patterns : elements of reusable object-oriented software. *Addison-Wesley,* 2003.

[11] **A. Gargantini, C. Heitmeyer.** Using model checking to generate tests from requirements specifications. In: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on F oundations of software engineering1999, Springer-Verlag: Toulouse, France. pp. 146-162.

[12] **A. Gotlieb, B. Botella, M. Rueher.** Automatic test data generation using constraint solving techniques, In: *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis* 1998, ACM Press: Clearwater Beach, Florida, United States. 53-62.

[13] **A. Gotlieb, T. Denmat, B. Botella.** Goal-oriented test data generation for programs with pointer variables. *29th Annual International Computer Software and Applications Conference (COMPSAC'05),* 2005, Vol. 442, pp. 449-454.

[14] **N. Gupta, A. P. Mathur, M. L. Soffa.** Automated test data generation using an iterative relaxation method, In: *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering* 1998, ACM Press: Lake Buena Vista, Florida, United States, pp. 231-244.

[15] **B. Hailpern, P. Santhanam.** Software debugging, testing, and verification. *IBM Systems Journal*, 2002, Vol. 41, No. 1, pp. 4-12.

[16] **H. Harmanani, B. Karablieh.** A hybrid distributed test generation method using deterministic and genetic algorithms. In: *Fifth International Workshop on System-on-Chip for Real-Time Applications (IWSOC'05)* 2005, pp. 317-322.

[17] **A. Hessel, P. Pettersson.** Model-Based Testing of a WAP Gateway: an Industrial Study. *FMICS and PDMC,* 2006.

[18] **R. Hierons, M. Harman, S. Danicic.** Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 1999, Vol. 9, No. 4, pp. 233-262.

[19] **P. Kalpana, K. Gunavathi.** A novel specification based test pattern generation using genetic algorithm and wavelets. 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05) 2005, pp. 504-507.

[20] **S. K. Kim, L. Wildman, R. Duke.** A UML approach to the generation of test sequences for Java-based concurrent systems. In: *2005 Australian Software Engineering Conference (ASWEC'05)* 2005, pp. 100-109.

[21] **B. Korel.** Automated test data generation for programs with procedures. In: *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*1996, ACM Press: San Diego, California, United States, pp. 209-215.

[22] **B. Korel, A. M. Al-Yami.** Assertion-oriented automated test data generation. In: *Proceedings of the 18th international conference on Software engineering*1996, IEEE Computer Society: Berlin, Germany, pp. 71-80.

[23] **P. Louridas.** JUnit: unit testing and coding in tandem. *IEEE Software*, 2005, Vol. 22, No. 4, pp. 12-15.

[24] **S. J. Mellor, A. N. Clark, T. Futagami.** Model-driven development - Guest editor's introduction. *IEEE Software*, 2003, Vol. 20, No. 5, pp. 14-18.

[25] **B. Meyer.** Object-oriented software construction. *Prentice-Hall, Inc.,* 1997.

[26] **A. Milicevic, S. Misailovic, D. Marinov, S. Khurshid.** Korat: A Tool for Generating Structurally Complex Test Inputs. In: *Proceedings of the 29th international conference on Software Engineering*2007, *IEEE Computer Society*, pp. 771-774.

[27] **E. M. Olimpiew, H. Gomaa.** Model-based testing for applications derived from software product lines. In: *Proceedings of the first international workshop on Advances in model-based testing 2005, ACM Press: St. Louis, Missouri,* pp. 1-7.

[28] **C. Oriat.** Jartege: a tool for random generation of unit tests for java classes. In: *First international conference on Quality of Software Architectures and Software Quality*, 2005. Erfurt, Germany: Springer-Verlag. pp. 242-256.

[29] **C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball.** Feedback-Directed Random Test Generation, In: *Proceedings of the 29th international conference on Software Engineering* 2007, IEEE Computer Society. pp. 75-84.

[30] **Š. Packevičius, A. Kazla, H. Pranevičius.** Extension of PLA Specification for Dynamic System Formalization. *Information Technology and Control*, 2006, Vol. 3, No. 36, pp. 235-242.

[31] **A. Paradkar.** Case studies on fault detection effectiveness of model based test generation techniques. In: *Proceedings of the first international workshop on Advances in model-based testing* 2005, ACM Press: St. Louis, Missouri, pp. 1-7.

[32] **[32] G. Patrice, K. Nils, S. Koushik.** DART: directed automated random testing. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* 2005, ACM Press: Chicago, IL, USA, pp. 213-223.

[33] **A. Seesing, H.-G. Gross.** A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. *International Transactions on System Science and Applications*, 2006, Vol. 1, No. 2, pp. 127-134.

[34] **J. M. Spivey.** Understanding Z: A Specification Language and Its Formal Semantics. *Cambridge University Press,* 2008.

[35] **N. T. Sy, Y. Deville.** Consistency techniques for interprocedural test data generation. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering 2003*, ACM Press: Helsinki, Finland. 108-117.

[36] **A. Uhl.** Model driven arcitecture is ready for prime time. *Software, IEEE*, 2003, Vol. 20, No. 5, pp. 70- 72.

[37] **A. Ušaniov, K. Motiejūnas.** A Method for Automated Testing of Software Interface. *Information Technology and Control*, 2011, Vol. 40, No. 2, pp. 99-109.

[38] **W. Visser, C. S. Pasareanu, S. Khurshid.** Test input generation with java PathFinder. In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*2004, ACM Press: Boston, Massachusetts, USA, pp. 97-107.

[39] **W. Xin, C. Zhi, L. Q. Shuhao.** An optimized method for automatic test oracle generation from real-time specification. In: *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)* 2005, pp. 440-449.

[40] **J. U. Zimmermann.** Informational integration of product development software in the automotive industry: the ULEO approach, 2005, University of Twente: Enschede, p. 245.

Received June 2012.