Journal of Engineering Science and Technology
Vol. 10, No. 3 (2015) 307 - 321
© School of Engineering, Taylor's University

EARLY IDENTIFICATION OF SOFTWARE DEFECTS USING OCL PREDICATES TO IMPROVE SOFTWARE QUALITY

A. JALILA*, D. J. MALA, M. ESWARAN

Department of Computer Applications,
Thiagarajar College of Engineering, Madurai, Tamil Nadu, India
*Corresponding Author: mejalila@gmail.com

Abstract

Formal specification-based testing has been used widely to assess potential faults or prove their absence in a given system at the earliest. This research work has proposed an automated fault-based testing framework to test the specification of the system using Object Constraint Language (OCL). Accordingly, the possible faults in OCL specification of the system has been anticipated by mutating its method based on OCL predicate-based fault classes. Then, test cases are generated using Genetic Algorithm with simulated annealing technique. In this paper, a novel OCL-predicate based fitness function is defined to evaluate the generated test data. Finally, this paper presents the experimental results, which indicate that the proposed methodology provides more test coverage with the reduced test suite and test run. This results in cost-effective software development so as to improve software quality.

Keywords: Specification-based testing, OCL, Genetic algorithm, Fitness function.

1. Introduction

Testing is one of the major quality criteria to guarantee the reliability of a system. However, testing activity will be more effective when it is applied from the initial stages of software development. Therefore, there is a great demand for specification-based functional testing to improve software quality. Fault-based testing is the most prominent form of specification-based testing [1]. It assumes that certain types of faults may be committed by programmers during software development, and based on that, various classes of faults are identified. Nevertheless, mutation testing is a kind of fault-based testing technique in which various classes of faults are injected into the SUT (System under Test) and test cases are designed to focus on those faults [2]. The reasoning behind this

approach is that a test suite which can detect seeded faults will also be sufficient to find the original faults in the specification.

Many prior research works [3-5] indicate that OCL has found its major application in precise software modelling [6]. However, the proposed approach is focus not only on the efficient system specification but also on the detection of specification-oriented faults at the earliest. Hence, delayed fault and failure detection increases the cost of error correction and maintenance [7]. Hence, in this research work, OCL predicate-based mutation test has been performed. The proposed approach endeavours to state that most of the abstract OCL specifications are complex predicates and not suitable for testing. Thus, in our approach, abstract OCL predicates are refined into equivalent simple Boolean expressions for effective testing. Then, mutants are automatically generated for all methods of the SUT by applying five fault classes which was proposed by Tai et al. [8]. Test paths are generated based on the coupling value which can be derived from OCL expressions of the SUT.

The proposed algorithm has used search-based technique, named, Genetic Algorithm (GA) in combination with the optimisation technique, namely, Simulated Annealing. This technique is used to select population, based on the fitness value. Based on this approach, unit testing is performed for each component of the SUT. In this paper, test adequacy criteria based on branch coverage and mutation scores are used to evaluate the fitness functions. For the purpose of this work, the terms specification, conditions, constraints and expressions are used interchangeably. On the same lines, specification-based testing would also mean mutation testing or fault-based testing. In this paper, SUT would mean the OCL specification of the system which is derived from its requirements specification. Similarly, component would mean class of the SUT.

The remainder of this paper is organised into the following sections. Section 2 starts with related earlier works. The background of the proposed OCL specification based testing is elaborated in Section 3. Section 4 discusses the proposed work. Section 5 deals experimentation of the proposed algorithm with various system specifications. The comparative study is detailed in Section 6 and conclusions are discussed in Section 7.

2. Related Work

Earlier studies show that there are various techniques available for the generation of test cases from OCL specification of the system.

Brucker et al. [9] proposed a theorem-prover based test data generation from OCL specifications. In their research, they had included language features such as recursive query-operations, object graph and equivalent relation. They have translated OCL Invariants into Recursive HOL-Predicates. However, in the proposed approach, OCL constraints are translated into equivalent simple Boolean-Expression predicates, for improved reasoning.

Benattou et al. [10] described the partition analysis technique to identify the domains of the operation of a specification. In their study, they used mathematics inherent which is derived from various constraints of the OCL specification to

generate test data. They reduced the mathematical expression of the OCL specification by transforming it into Disjunctive Normal Form (DNF).

Bao-Lin et al. [11] adopted a methodology to generate test cases based on the combination of UML sequence diagram and OCL. In their approach, first they constructed a tree based on the sequence diagrams of a system. The resultant tree was traversed and conditional predicates are selected from the sequence diagram. Then, relevant OCL constraints were specified to generate test cases. Message paths coverage and constraint attribute coverage were used to assess the test adequacy criteria. However, in the proposed algorithm, mutation score and branch coverage based test adequacy criterion are used as the fitness functions

Ali et al. [12] devised a methodology to generate test data from OCL constraints which are derived from UML state and class diagrams. Their study used search-based techniques, such as Genetic Algorithms and Evolutionary Algorithm to derive test data from OCL constraints. They defined a set of heuristics based on OCL constraints and branch distance functions for various types of OCL expressions.

Aichernig and Salas [13] implemented mutation testing approach for automatic test case generation from model-based specifications. In their work, anticipated faults were generated from OCL. Then, test cases were generated to discover such faults.

From the existing research works, it has been inferred that the implementation details for automatic test generation and execution using OCL are not either explained clearly or they do not exist in the earlier works. Most of the prior works have not discussed the OCL collection operations. With respect to the existing literature, the major objective of this study is to propose a methodology for automatic test generation using OCL in which all collection operations are refined into alternative decision statements. Moreover, the present study suggests that OCL predicate with search algorithm ensures effective functional testing.

3. Background

This section describes the basic concepts involved in OCL specification-based testing, which are detailed as follows.

3.1. Object constraint language (OCL)

OCL stands for Object Constraint Language [14]. It was proposed by the Object Management Group (OMG). OCL defines the system in the form of constraints or predicates. It is based on the simple mathematical notations or predicate logic that enables rigorous analysis and reasoning about the specifications. In general, OCL is used for precise model specification.

In the proposed approach OCL constraints have been derived from user requirement specification of the system. There are two major building blocks of OCL, that are method definition (which includes pre-conditions and post-conditions definition) and invariant (condition) declaration. The pre-condition is a Boolean expression that evaluates to true before the execution of a method. The

post-condition is a Boolean expression that evaluates to true after the execution of a method. The invariant is a Boolean expression that evaluates to true always. OCL-based testing is preferable due to the following reasons:

- OCL is platform-independent, and hence, it is applicable to all types of systems. Moreover, it describes precise model of a system; hence, functional test results will be accurate.
- OCL constraints can be easily derived from requirements specification of the system and the changes in specification do not imply extensive re-writing of the test-code.
- Supports functional testing at the early stages of software development and also the test coverage is improved because the coverage is measured from the specification.
- Specification refinement of OCL predicate requires less effort and test paths can be easily derived from OCL expressions.

3.2. Genetic algorithm (GA) with simulated annealing (SA)

GA is selected for the proposed approach hence, it is the most commonly used search algorithm in search-based software engineering [15]. There are three major operators used in GA, namely selection, crossover and mutation. However, GA suffers from problems like slow convergence, blind search and risk of getting stuck into local optimum solution [16]. Hence, the proposed approach combines the strength of the simulated annealing algorithm to produce efficient test cases.

3.2.1. Simulated annealing

Simulated annealing is a memory-based adaptive algorithm that guides local heuristic search procedure to explore solution space beyond local optimality. The basic principles of the SA include: A method to generate initial configuration; transition or generation function to find a neighbour as next candidate; cost function; evaluation criterion and stop criterion. In order to reduce the randomness and execution time of GA based search, simulated annealing search heuristic can be applied during the mutation step. However, the candidates will be accepted only if fitness value is higher than current configuration. This algorithm will repeat until no neighbour with higher fitness is found.

3.2.2. Pseudo-Code of GA with SA

Step 1: Initialise the temperature T and N, the number of chromosomes by generating N possible solution randomly.
Step 2: Find the fitness value for each and every chromosome using branch coverage and mutation score.
Step 3: Select N/2 chromosomes using the Stochastic Universal Sampling (SUS) from N chromosomes.
Step 4: Crossover and Mutate the selected chromosomes to get new chromosomes.
Step 5: Find the fitness values for newly generated chromosomes.
Step 6: Choose the N best chromosomes which have the maximum fitness

values compared to the newly generated as well as old chromosomes.

Step 7: Find the best among the N chromosomes.

Step 8: If the best chromosome is not changed over a period of time then find a new chromosome

Step 9: Accept the new chromosome as the best one with probability as exp-(ΔE/T), where, ΔE is the difference between the current best chromosome's fitness and the new chromosome's fitness value.

Step 10: Reduce the temperature and repeat until either the maximum number of iterations is reached or optimal value is obtained.

3.2.3. Test case generation using GA with SA

The example for the GA with SA algorithm for test data generation is detailed below:

Initial Temperature 100

Crossover	**Mutation**
Test Case1: {1000,"xxxx","FD", 1599}	Mutation point: 4
Test Case2: {1010,"yyyy","CUR", 10009}	Test Case: {1000,"xxxx","FD", 10}
Crossover point: 2	
Test Case1: {1000,"xxxx","CUR", 10009}	Test Case: {1000,"xxxx","FD", 31}
Test Case2: {1010,"yyyy","FD", 1599}	

Evaluation:

E= fitness (test case1) = 36%

E'=fitness (test case2) = 88%

ΔE= E'-E

If(ΔE<0) then test case2 will be selected

Else if (exp-(ΔE/T)> rand (0, 1)) then test case2 will be selected

Else No change End if End if

Result: Test case 2 is selected

4. Proposed Work

The proposed automatic OCL specification-based testing framework as illustrated in Fig. 1 has the six major modules.

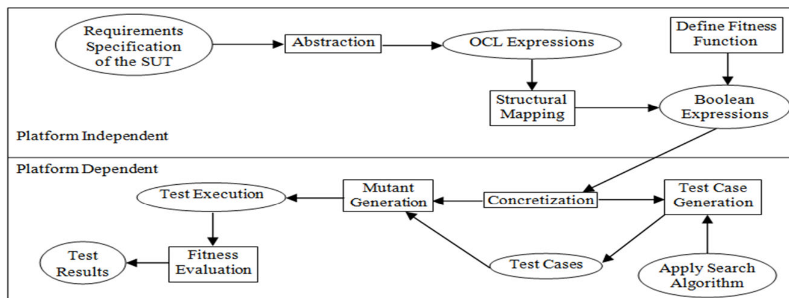
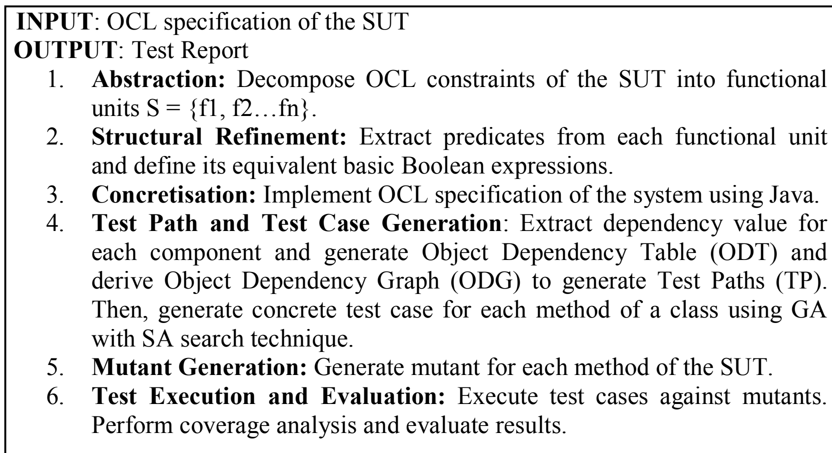


Fig. 1. Proposed OCL-based functional testing framework.

Abstraction: Formalisation of user requirement specification of a system using OCL. **Structural Mapping**: The abstract model of the system must be

precise and complete enough to derive tests from them. Hence, OCL predicates are transformed into equivalent basic Boolean expressions. Then, abstract test cases are derived from it. Furthermore, fitness function is defined to estimate test adequacy criteria. **Concretisation:** In order to automatically generate and execute test cases, the OCL specification of a system is converted into code using implementation langue such as Java, C++ etc. **Test Path and Test Data Generation**: Test paths are derived by extracting dependency value of each component of the system then concrete test cases are generated for each method of a class using by applying search algorithms. **Mutant Generation**: Mutants are generated for each method of a class based on the appropriate fault classes. **Test Execution and Evaluation**: The generated test cases are executed against mutants of the SUT and test results are evaluated based on the fitness value.

Figure 2 presents the proposed algorithm to perform functional testing using OCL. The following section details some of the concept used in the proposed algorithm.



INPUT: OCL specification of the SUT
OUTPUT: Test Report

1. **Abstraction:** Decompose OCL constraints of the SUT into functional units S = {f1, f2…fn}.
2. **Structural Refinement:** Extract predicates from each functional unit and define its equivalent basic Boolean expressions.
3. **Concretisation:** Implement OCL specification of the system using Java.
4. **Test Path and Test Case Generation**: Extract dependency value for each component and generate Object Dependency Table (ODT) and derive Object Dependency Graph (ODG) to generate Test Paths (TP). Then, generate concrete test case for each method of a class using GA with SA search technique.
5. **Mutant Generation:** Generate mutant for each method of the SUT.
6. **Test Execution and Evaluation:** Execute test cases against mutants. Perform coverage analysis and evaluate results.

Fig. 2. Proposed OCL-based functional testing algorithm.

4.1. OCL-Structural mapping

The specifications written in OCL are essentially in textual form which is non-executable. OCL provides a special kind of data type, namely OclAny which includes two major types such as primitive and collection data types. Primitive types include Integer, Real, String and Boolean [17]. The collection data types include Set, OrderedSet, Sequence and Bag. Moreover, OCL provides a large number of operations on collection data types.

However, Boolean operations such as 'implies' and the collection operations are exclusive to OCL expressions and do not appear in implementation languages such as C++, Java etc. Thus, automatic functional testing directly from abstract OCL specification would be ineffective. Thus, in order to achieve rigorous function testing on OCL specification, all the predicates defined in abstract OCL specification are refined into executable Boolean expressions. It is formally defined as follows.

Definition 1: Structural mapping is defined as a function $[|\ |]^E$ which translates each part of abstract OCL expression (S_A) into its counterpart executable expression(S_E). Hence, the behaviours of S_A include all those of S_E. Therefore, S_A is equivalent to S_E. Figure 3 illustartes the strutural mapping function. It is formally represented as

$$[|\ |]^{\mathbf{E}}: S_A \sqsubseteq S_E \quad \text{or} \quad [|\ |]^{\mathbf{E}}: S_A \rightarrow S_E \tag{1}$$

The structural mapping functions for the four OCL elements, namely numeric and Boolean type operations, Standard and Iterative OCL collection Operations are detailed as follows.

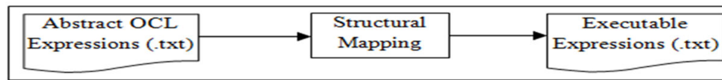


Fig. 3. Stuctual mapping function.

4.1.1. Numeric and Boolean type operations

There are two types of operators, namely relational and arithmetic defined for the OCL numeric data types such as Integer and Real. The arithmetic functions such as abs(), div(), mod(), max(), and min() are used as part of the arithmetic calculation with arithmetic operators. Hence, the structural mapping for the arithmetic function are common, direct hence, they are not defined in this section. The proposed work endeavours that if an OCL expression contains relational or Boolean type operators the notation for the abstract OCL expressions are transformed into simple branch statements. Table 1 describes the structural mapping function for the relational and Boolean type operations. It is formally represented as follows

$$[|\text{Relational Operations}|] \text{ or } [|\text{Boolean Operations}|] \rightarrow [|\text{Branch Statement}|]^E \tag{2}$$

Table 1. Structural mapping for the relational and Boolean operations.

Relational Operations (S_A)	Structural Mapping (S_E)	Boolean Operations (S_A)	Structural Mapping (S_E)
a <> b	[|if a!=b then return True|]^E	**a**	[|if a=True then return flag|]^E
a > b	[|if a > b then return True|]^E	**a and b**	[|if a and b=True then return flag|]^E
a ≥ b	[|if a≥b then return True|]^E	**a or b**	[|if a or b=True then return flag|]^E
a ≤ b	[|if a ≤b then return True|]^E	**a** implies **b**	[|If a=True then b=True |]^E
a <b	[|if a <b then return True|]^E	**a xor b**	[|If a=True then b=False|]^E
a=b	[|if a =b then return True|]^E	**If a then b else c**	[|if (exp a)=True then return b else return c|]^E
a <> b	[|if a! =b then return True|]^E		

4.1.2. Standard OCL collection operations

There are several standard collection operations are available in OCL. Table 2 depicts the structural mapping for the standard OCL collection operations. The

collection operations namely includes() and excludes() are used to check the existence or absence of a particular object in a collection respectively.

The collection operation such as includesAll() ensures whether all the elements of a component contained in the collection or not. Its inverse is excludesAll(). The availability of elements in a collection can be verified using the two collection operations, namely isEmpty() and notEmpty().

4.1.3. Iterative OCL collection operations

The iterative collection operation such as isUnique() determines whether a collection has unique value for a particular attribute or not exists() operation is used to check whether a Boolean expression has at least one element for which

Table 2. Structural mapping of the standard OCL collection operations.

Standard OCL Operations (S_A)	Structural Mapping (S_E)
a→includes(b): Boolean	[|loop: i=1 to b.size() if b[i]!=null then return True |] ^{E}
a →excludes(b): Boolean	[|loop: i=1 to b.size() if b[i]=null then return True |]^E
a→includesAll (b:T):Boolean (b:Collection(T)): Boolean	[|loop: i=1 to a.size(), loop: i=1 to b.size() if a[i]= b[i] ^{E} then return True |] ^{E}
a→ excludesAll(b:T):Boolean Boolean	[|loop: i=1 to a.size(), loop: i=1 to b.size() if a[i]!= b[i] ^{E} then return True |] ^{E}
isEmpty():Boolean	[|if a. size()!=0 then return True |] ^{E}
notEmpty():Boolean	[|if a. size()!=0 then return True |] ^{E}

the predicate ‘P' is true. The operation forAll() determines whether the expression is true for all elements in the collection. select() operation returns all elements in a collection for which ‘P’ is true and reject() returns all elements in a collection for which ‘P’ is false. Collect and iterator operations used to choose a subset of elements in a collection whereas reject selects all elements of a collection for which a Boolean expression is false. Table 3 depicts the structural mapping function for the iterative OCL collection operations.

Table 3. Structural mapping of iterative OCL collection operations.

Collection Operations (S_A)	Structural Mapping (S_E)
isUnique(b:T): Boolean	[|loop a=1 to a.size()-1 if a[i]!=a[i]+1 then return True |]^E
c→ exists(b | P): Boolean, where p is the predicate	[|loop i=1 to a.size(), if a[i]=b then return True|] ^{E}
a→ forAll(v1, v2…vn: T| p): Boolean, where	[|loop i=1 to a.size(),p[a[i]|] ^{E}
a → select(b:T): Boolean, where b is any object	[|loop i=1 to a.size(), if exp[a[i])=True then return Collection|]
a → reject(b:T): Boolean, where b is any object	[|loop i=1 to a.size(),if exp[a[i])!=True then Collection[i]=a[i] return Collection|]

The string operations such as size(), concat(), substring(), toInteger() and toReal() are available in all implementation languages. Hence, the structural

mapping function is not required for the string operations. The other OCL miscellaneous operation such as @pre (denotes the value of a attribute before the start of the method), append(), count() and size can be easily mapped into executable statements. Hence, those simple operations are not discussed in this paper.

4.2. Fault classes for OCL predicate testing

There are different types of faults that can occur in the software or specification of the system. Moreover, exhaustive testing requires more time and lager memory space. Hence, the fault-based testing approach hypothesises certain types of faults which frequently occur in the specification or program of a system. Those hypothesised faults are named as fault classes. Each fault class is designed to detect a particular type of fault. Reasoning behind this approach is coupling effect [18], that is, if a set of test cases detects a lot of simple errors, then it would also be sufficient to detect larger and more complicated faults.

The cost of testing can be affected by the choice of fault class. Hence, there are five predicate-based fault classes as proposed by Tai et al. [8]. Table 4 furnished the details of the fault classes which have been used in the proposed approach to generate mutants.

Table 4. Tai's predicate-based fault classes.

Fault Class	Description	Mutant Generation	
		Original Predicate	Mutant
BOOF	Incorrect AND/OR or Missing/Extra NOT Operator	a !=b	a =b
IROF	Incorrect Relational Operator	a >b	a<b
ILPF	Incorrect Location of a Parenthesis	a→ size()=0	a→ size(=0)
IBVF	Incorrect Boolean Variable	x = True	y = True
IAEF	Replacement of Arithmetic Operators	x = a + b	x = a - b

4.3. Dependency value estimation

The dependency value points out the non-inheritance based relation for a class. A class can reference another class by accessing its attribute or method or instance. Let us consider a class 'C' references the other classes such as j= {0…n}, then the dependency value (DV) of a component 'C' is given by

$$\mathrm{DV}(\mathrm{C}) = \sum_{j=0}^{n} C_j(\mathrm{C}) \quad (3)$$

The dependency value will be stored in ODT. Based on the ODT values, the ODG is constructed by considering each component as a node (with flag value) and the dependency among them as edges. Then, ODG is traversed in depth first manner to generate test paths.

4.4. Fitness function

The generated test cases are executed against both the originals and mutants. Based on the outcome, the method-wise mutation score and the branch coverage value are calculated for evaluating the test case adequacy criteria.

Method-wise Mutation Score: If a class 'C' contains methods m = {1…n}, where, F_m is the number of methods exercised by the test case T, C_m is the total number of methods defined for the component 'C'. Then, the method-wise mutation score (MS) against test case (T) is given by

$$MS_m(T) = \frac{|Fm|}{|Cm|} \times 100 \quad (4)$$

Branch Coverage: Most of the executable form of OCL pre-conditions, post-conditions and invariants are branch statements. If executable expression of a class 'C' contains branch statements bs = {1…n}, N_{bs} is the number of branches statements in a class 'C' which are covered using test case (T), T_{bs} is the total number of branches in class 'C'. Then the branch coverage value (BCV) against test case (T) is given by

$$BCV_{bs}(T) = \frac{|Nbs|}{|Tbs|} \times 100 \quad (5)$$

5. Experimentation and Result Analysis

This section briefs the implementation of the proposed algorithm. The proposed algorithm is coded in Java. The algorithm is tested with OCL specifications of various real time applications.

5.1. Experimental setup

For the experimental purpose of this research work, the OCL specification of the seven industrial applications has been selected. Table 5 describes the brief summary of these case study applications.

Table 5. Brief summary of the case study applications.

Application Name	App-ID
Patient Monitoring	PMS
Blood Bank Management	BBMS
Library Management	LMS
Payroll Management	PAYROLL
Banking Management	BMS

Each specification varies in the number of classes and the complexity of the model. These case study applications are relatively small compared to many real time application specifications, but are logically complex enough for the purpose of the proposed work.

5.2. Experimentation

This section explains the OCL-based test case generation using GA with SA. The application selected for this study is PAYROLL system.

Figure 4 presents the sample abstract of OCL expressions for the 'admin' class in the PAYROLL system. There are three functional units defined for the 'admin' class, namely 'calculate-salary', 'update-emp' and 'calculate-attendance'.

```
context admin::calculate-salary(empid:Integer,s:salary,
ad:adjustment,a:attendance,amount:Real):Real
pre:empid>0
pre: s.basicsal> 3000
pre: a.noofpre<> 0
post:s.gp= (s.basicsal/ 30)*(a.noofpre + a.noofhol)-ad.adjust
post:ad.adjust>s.totalsal
post:result= s.gp - ad.adjust

context admin::update-emp(e:employee)
pre: employee->excludes(e)
post: employee ->includes(e)

context admin::calculate-attendance(empid:Integer,a:attendance)
pre:empid>0
pre:a.workday>1
pre:a.noofhol>=0
post: a.noofpre<=self.workday--B2
post: a.noofpre=(a.workday + a.noofhol
post:a.noofpre=(a.workday-a.noofpre) + a.noofhol
```

Fig. 4. Abstraction of the PAYROLL system.

Figure 5 depicts the equivalent structural mapping for the constraints defined for the method ‘calculate-attendance’ in class ‘admin’ of the PAYROLL system. The executable OCL expressions of the system are implemented using Java.

```
Class admin
Calculate calculate-attendance(a:attendance)
workday(attendance)>1
noofhol (attendance)>=0
noofpre(attendance)<=workday(attendance)
noofpre=workday(attendance)+noofhol(attendance)
noofpre=workday(attendance)-noofpre(attendance)+
noofhol(attendance)
```

Fig. 5. Sample for structural mapping.

The left hand side of the Table 6 presents the equivalent Java code for the method ‘calculate-attendance’ of the ‘admin’ class in the PAYROLL system.

In order to generate test paths, the ODG of the system is constructed by extracting coupling value of each component of the SUT from its OCL expressions that will be stored in the ODT. Table 7 depicts the ODT values for the PAYROLL system.

Based on the ODT values, the ODG is constructed by considering each component as a node (with flag value) and the dependency among them as edges. Figure 6 shows the object dependency graph for the PAYROLL system. In the proposed algorithm, the ODG graph is traversed in depth first manner to generate test paths. The following are three paths generated for the PAYROLL system based on its ODG which is given in Fig. 6. Here, TRP represents the test path requirement. The generated test paths cover all the nodes and edges in ODG of the PAYROLL system.

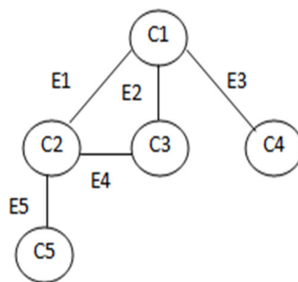
Table 6. Mutants generation.

OCL Expressions (Equivalent Java program)	Mutant
public class attendance {	public class attendance {
public void calculate-attendance(int present, int workdays, int holidays) {	public void calculate-attendance (int present, int workdays, int holidays) {
int attend=0;	int attend=0;
attendance a= new attendance()	attendance a= new attendance()
a.present = present	a.present = present
a.workday = workdays	a.workday = workdays
a. holiday= holidays	a. holiday= holidays
if (a.workday >=18 and	if (**a.workday <=18** and
a.holiday<=8) { //B1	a.holiday<=8) { //B1
if (a.present>=a.workday) { //B2	if (**a.present<=a.workday**) { //B2
a.attend= a.workday +a.holiday	a.attend= **a.workday - a.holiday**
else // B3	else // B3
a.attend= a.present+ a.holiday }}}}	a.attend= **a.present- a.holiday** }}}}

*B1, B2, B3 are branches

Table 7. Object dependency for PAYROLL system.

Component	Method	Dependency (DCC)	Edges Name
C1: admin	C1: calsal	C2	E1
C1: admin	C1: calatt	C3	E2
C1: admin	C1: updateemp	C4	E3
C2: Salary	C2:salaryinfo	C3	E4
C3:attendance	C3:attendinfo	-	-
C4: employee	C4: empinfo	-	-
C5:PF	C5: PFcalc	C2	E5



TPR = {C1, C2, C3, C4, C5}
Test path 1: C1→ C2→C5
Test Path 2: C1→ C3 → C2→C5
Test Path 3: C1→ C4

Fig. 6. Test path generation.

As discussed in section 3.2.3, test cases are generated for each method of a class using GA with SA algorithm. Then, faults are introduced in each method of a class based on Tai et al. [8] fault classes as discussed in section 3.4. The right hand side of the Table 6 presents the sample mutants generated for the method ‘calculate-attendance’ in the PAYROLL system.

The generated test cases are executed against mutants. Then, the capability of the test case is analysed based on its ability to distinguish the original expressions with the mutant as is illustrated in Fig. 7.

```
Initial set of Test cases
 Testcase1: 18,2,18
 Testcase2: 18,0,16

 Brach 1 is covered with the test case 18,2,18
 Branch coverage is 33%
 Method wise score for the test case18,2,18 is 100%

 Brach 1 and 3 is covered with the test case 18,0,16
 Branch coverage is 67%
 Method wise score for the test case 18,0,16 is 100%
```

Fig. 7. Sample for test execution and evaluation.

6. Comparative Study

This section presents the result of experiments that were carried out to evaluate the effectiveness of the proposed OCL-predicate based testing. The test objects have been tested using the three techniques, namely random, simple GA and the proposed GA with SA algorithm. The results were gathered in terms of their branch coverage (BC) in percentage (%), method-wise mutation score (MS) in %, and test cases (TC) generated in numbers as shown in Table 8.

From Table 8 it can be inferred that the proposed algorithm adopting GA with SA algorithm provides much better results in terms of high coverage and minimal number of test cases in all the sample applications.

Table 8. Results of the test problems.

S. No	Case Study ID	Random Search			GA			GA with SA		
		BC %	MS %	No. of TC	BC %	MS %	No. of TC	BC %	MS %	No. of TC
1	**PMS**	72	75	858	93	75	231	97	85	133
2	**BBMS**	68	54	758	95	82	354	98	92	248
3	**LMS**	71	57	986	91	68	311	96	89	205
4	**PAYROLL**	74	74	1758	96	86	622	99	94	483
5	**BMS**	78	58	1859	93	79	733	98	92	565

7. Conclusion

In this research work, a novel approach has been adopted to automate functional testing based on OCL predicates. In this approach, test cases have been generated using two search techniques namely GA along with SA. The proposed approach can generate highly efficient test cases at the specification level. The major merits of the proposed framework are: structural mapping, dependency value extraction,

test path generation, test case generation using GA with SA algorithm and fitness evaluation based on mutation score and branch coverage.

From the above study, it has been inferred that OCL specification-based testing is essential to improve software quality that supports effective testing with more test coverage. The proposed approach has been applied to many real-time applications and observed the effectiveness of specification based testing using OCL predicates.

The proposed approach has been found to generate test cases efficiently, thereby improving the overall software quality. As a future work, it has been proposed to adopt other optimisation techniques such as Ant Colony Optimisation (ACO), Harmony Search etc., to minimise the test requirements with less test runs.

Acknowledgment

This paper is a part of the UGC major research project supported by University Grants Commission (UGC), New Delhi, India.

References

1. Richard, K.D. (1999). Fault classes and error detection capability of specification based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4), 411-424.
2. Morell, L.J. (1990). A theory of fault-Based testing. *IEEE Transactions on Software Engineering*, 16(8), 844-857.
3. Agustín, G.; and Yadran, E. (2004). Building precise UML constructs to model concurrency using OCL. *Lecture Notes in Computer Science,* 3273, 212-225.
4. Klasse (2008). Octopus: OCL tool for precise UML specifications. Retrieved October 5, 2008 from http://www.klasse.nl/octopus/ index.
5. Warmer, J.; and Kleppe, A. (2003). *The object constraint language: Getting your models ready for MDA*. Addison-Wesley.
6. Cabot, J.; and Gogolla, M. (2012). Object constraint language (OCL): A definitive guide formal methods for model-driven engineering. *Lecture Notes in Computer Science*, 7320, 58-90.
7. Elberzhager, F.; Rosbach, A.; Münch, J.; and Eschbach, R. (2012). Reducing test effort: a systematic mapping study on existing approaches. *Information and Software Technology*, 54(10), 1092-1106.
8. Tai, K.C.; Vouk, M.A.; Paradkar, A.M.; and Lu, P. (1994). Evaluation of a predicate-based software testing strategy. *IBM Systems Journal*, 33(3), 445-457.
9. Brucker, A.D; Krieger, M.P; Longuet, D.; and Wolff, B. (2010). A specification-based test case generation method for UML/OCL. *Lecture Notes in Computer Science*, 6627, 334-348.
10. Benattou, M.; Bruel, J.M.; and Hameurlain, N. (2002). Generating test data from OCL specification. *Proceedings of the ECOOP'2002 Work-Shop on Integration and Transformation of UML Models.*

11. Bao-Lin, L.; Zhi-shu, L.; Qing, L.; and Hong, C.Y. (2007). Test case automate generation from UML sequence diagram and OCL expression. *International Conference on Computational Intelligence and Security*, 1048-1052.
12. Ali, S.; Iqbal, M.Z.; Arcuri, A.; and Briand, L. (2013). Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10), 1376-1402.
13. Antonio, P.; Salas, P.; and Aichernig, B.K. (2006). Automatic test data generation for OCL: A mutation approach, *Proceedings of 5th International Conference on Quality Software, IEEE Computer Society*, 64-71.
14. Object Constraint Language Specification (2010). Object management group (OMG), Version 2.2. Retrieved April 5, 2010 from http://www.omg.org/spec/CL/2.2/.
15. Harman. (2007). The current state and future of search based software engineering. *Proceedings of the International Conference on Future of Software Engineering (FOSE'07*), IEEE Press, 342-357.
16. Shen, X.; Wang, Q.; Wang, P.; and Zhou, B. (2009). Automatic generation of test case based on GATS algorithm. *IEEE International Conference on Granular Computing*, 496-500.
17. Philippe, C.; and Roger, R. (1999). Towards efficient support for executing the object constraint language. *TOOLS '99 Technology of Object-Oriented Languages and Systems, IEEE Computer Society Washington*, DC, USA, 399-408.
18. Offutt, A.J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1), 5-20.