DOI: 10.21917/ijsc.2015.0152

OCL-BASED TEST CASE GENERATION USING CATEGORY PARTITIONING METHOD

A. Jalila[1] and D. Jeya Mala[2]
Department of Computer Applications, Thiagarajar College of Engineering, India
E-mail: [1]mejalila@gmail.com, [2]djmcse@tce.edu

Abstract

The adoption of fault detection techniques during initial stages of software development life cycle urges to improve reliability of a software product. Specification-based testing is one of the major criterions to detect faults in the requirement specification or design of a software system. However, due to the non-availability of implementation details, test case generation from formal specifications become a challenging task. As a novel approach, the proposed work presents a methodology to generate test cases from OCL (Object constraint Language) formal specification using Category Partitioning Method (CPM). The experiment results indicate that the proposed methodology is more effective in revealing specification based faults. Furthermore, it has been observed that OCL and CPM form an excellent combination for performing functional testing at the earliest to improve software quality with reduced cost.

Keywords:

Software Testing, Specification-based Testing, OCL, CPM, Formal Specification

1. INTRODUCTION

The fulfillment of requirement of the user is the major aim of software development. Improper specification and design leads to high cost in the implementation phase [1]. Therefore, software systems are formally specified using any one of the specification languages, namely OCL, Z, B, VBM, alloy.etc. The 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 testing in the field of formal specification based software development. Hence, formal specification of the system serves both as a test oracle and medium to generate functional test cases. It is a black box type testing and it is used to confirm whether each function of the system works based on the requirements or not. Specification based testing approach is used to develop test cases and perform coverage analysis in a simple and effective manner.

In particular, there is a wide acceptance for OCL in precise software specification. The OCL specification can be derived both from requirement specification (during the requirement specification phase) or from class diagram (during the design phase). OCL describes the prototyping of a system; hence, functional test results will be accurate. OCL provides strong base for system testing [2]. Hence, the proposed approach extended the application of OCL both for specification and design time testing. However, OCL is essentially a textual form of first order predicate logic [3] and non-executable language [4]. Therefore, the generation of test cases directly from OCL specification would be ineffective [5]. There are many prior works which have described the process of test generation from OCL using various techniques [6, 7, 8, 9, 10 and 11]. The proposed algorithm adopts existing specification-based testing techniques named Category Partition Method (CPM) to OCL. In our work, SUT would mean the requirements specification of the system.

The CPM was developed by Ostrand and Balcer [12] to generate test cases from functional specifications. Though it is a traditional approach, it is more effective in deriving functional test case generation. Moreover, it has not been used for OCL specification so far. The main purpose of this approach is to find all possible choices among constraints more effectively and generate test cases more efficiently.

The remainder of this paper is organized into the following sections. Section 2 describes the basic concepts for the proposed study. The related work is elaborated on in section 3. Section 4 deals the proposed algorithm. The experimentation of the proposed algorithm appears in section 5. The comparative study has been detailed in section 6 and section 7 has drawn conclusions out of this study.

2. BASIC CONCEPTS

2.1 OCL FORMAL SPECIFICATION

OCL standard stands for Object Constraint Language. It was proposed by OMG organization. OCL is a model based specification language. An UML diagram can not reflect all relevant aspects and constraints of a model. Thus, OCL standard has been developed to extend UML models by defining constraints.

OCL is based on the simple mathematical notations. There are three kinds of constraints or conditions which form the building blocks of OCL expressions, namely invariants, pre-conditions and post-conditions. To generate the test cases from OCL based specification, there is need to find out the dependent classes or the classes which require the service of the given class. To achieve this, dependent metric is extracted from OCL and is described in section 2.2.

2.2 DEPENDENCY METRICS EXTRACTION FROM OCL SPECIFICATION

The dependency between classes can be identified using direct class coupling (DCC) metric. DCC metric was proposed by Bansiya et al. [9]. The DCC metric value is extracted from OCL specification of the SUT based on the method parameter definition and invariant declaration.

Let there be a component namely *C*, containing a method *M*. There are two types of parameters can be passed to a method, namely parameter as reference, parameter as variable. The parameters of method M are expressed in OCL as follows

$$context\ C::M(v_1 : T_1, a_1: C_1, ..., v_n : T_n):RT \quad (1)$$

where, $V= \{v_1...v_n\}$ are the formal parameters of the method M. $T = \{T_1...T_n\}$ are the parameter types, a_1 object for the class C_1. RT is the return type for the method M.

From Eq.(1) it has been observed that component C is coupled with another component named C_1.

The invariant of the component C can be defined using OCL expressions as follows.

$$context\ C\ inv\ invName:C_t \quad (2)$$

or

$$context\ C\ inv\ invName:self.C_2.C_{2t} \rightarrow C_t \quad (3)$$

where, C_2 is the associated class name, C_{2t} is the associated class attribute and C_t is the condition as applicable to the class C. From Eq.(3) it has been observed that the component C is coupled with another component C_2.

2.3 CATEGORY PARTITIONING METHOD (CPM)

CPM is the methodology applied to generate test cases from formal specifications of the SUT [8]. The following are the steps involved in category partitioning method

Step 1: Decompose the OCL constraints of SUT into functional Units.

Step 2: Find parameters, pre and post conditions of each function unit.

Step 3: Identify categories for each parameter and environment conditions (pre and post conditions).

Step 4: Find the choices for each category by providing all different kinds of values that are possible for it.

Step 5: Generate test frame by establishing the constraints among the choices of different categories.

Step 6: Select a single element from each choice and derive test cases.

2.4 COVERAGE CRITERIA FOR OCL SPECIFICATION USING CPM

There are three coverage criteria are used in the proposed approach namely statement coverage, condition coverage, parameter coverage and path coverage.

Statement Coverage: It has been endeavored in the proposed work that statement coverage is the ratio between numbers of statements exercised to the total number of statements which are available in the OCL specification of the system.

$$\text{Statement Coverage} = \frac{\text{No. of statements excercised}}{\text{Total No. of statements}} \times 100 \quad (4)$$

Parameter coverage: It is the ratio between the total number of parameters exercised to the total number of parameters in the OCL specification of the system.

$$\text{Parameter coverage} = \frac{\text{No. of parameters excercised}}{\text{Total No. of parameters}} \times 100 \quad (5)$$

Condition coverage: It is the ratio between number of pre and post conditions exercised to the total number of pre and post conditions in the OCL specification of the system.

$$\text{Condition Coverage} = \frac{\text{No. of pre and post condtions excercised}}{\text{Total No. of pre and post conditions}} \times 100 \quad (6)$$

Path coverage: It is the ratio between the number of paths in the ODG exercised to the total number paths in the ODG of the system.

$$\text{Path Coverage} = \frac{\text{No. of test path excercised}}{\text{Total No. of testpaths}} \times 100 \quad (7)$$

3. RELATED WORK

According to Amla and Ammann [10] CPM can be applied to the functional specification of the natural-language. In their study, they used Z specifications and CPM to generate test cases, whereas in our proposed approach, OCL specification of the system is used as a base for test case generation.

Ammann and Offutt [11] devised a methodology to generate test script and coverage criteria, named base-choice-coverage, for Z specification. In their approach they used category-partition method-based testing. In this approach, statement, node and edge coverage criteria are used.

Offutt and Irvine [12] used mutation approach and CPM to test C++ programs. In their approach common type of faults in C++ programming are inserted to the programs. Then, test cases are generated using CPM to uncover seeded faults. In our proposed methodology depth first search technique is used to generate test paths.

Grochtmann and Grimm [13] proposed a classification tree method to construct test cases from functional specifications. Chen et al. [18] constructed test frames to improve the tree structure for supporting category-partition based test case generation.

The authors [15] used XML schema mapping and category partition to identify the related constraints and relevant values for each category.

Many earlier studies have demonstrated the test case generation procedure from OCL. However, the implementation details are either not covered or poorly explained. Thus, there is a need for a standard and simple procedure to generate test cases from OCL formal specification. The major difference between our approach and the previous research works is that, our work is based on OCL-specification based CPM. In addition different types of OCL-based coverage criteria are introduced in our approach.

4. PROPOSED WORK

The proposed work includes proposed frame work and algorithm which are detailed in section 4.1 and 4.2 respectively.

4.1 PROPOSED FRAMEWORK

The proposed framework consists of four modules, namely OCL constraint extractor, test path generator, test case generator and coverage analyzer which are depicted in Fig.1.

First, OCL specification of the system is derived which is given as input to the test path generator module. The dependencies between the components of the system are assessed by deriving Direct Class Coupling (DCC) metric from invariant and method parameter definitions of a component. In test case generator module, abstract test cases are generated from OCL constraints of the system using CPM. Then, test coverage is analyzed.

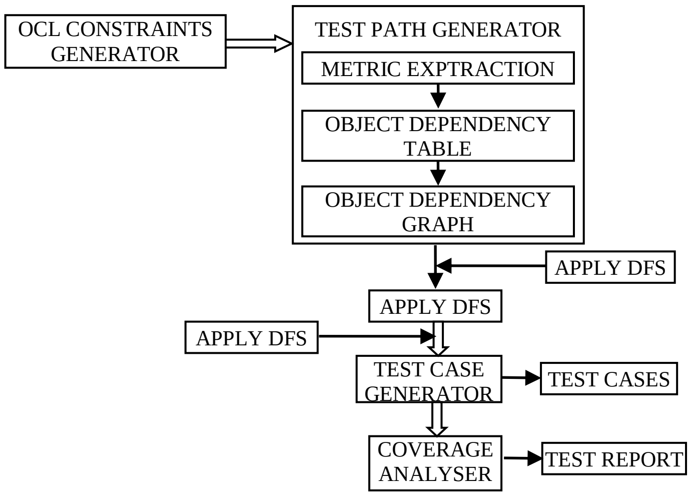


Fig.1. Proposed Framework

4.2 PROPOSED ALGORITHM

INPUT: OCL specification of the SUT

OUTPUT: Test suite TS.

Step 1: Derive OCL expressions of a system from requirement specification and save it with .txt file extension

Step 2: Generate Test path based on the object dependency table and dependency Graph

Step 2.1: Parse the .txt file using java to derive DCC metric and generate Object Dependency Table (ODT) and assign identifier for each object.

Step 2.2: Construct Object Dependency Graph (ODG) from the Object Dependency Table (ODT) where each node represents the method and label edges.

Step 2.3: Traverse the Object Dependency Graph (ODG) in Depth First Search (DFS) manner to derive test paths. Generate OCL-based test cases with CPM do the following steps

Step 3:

```
    $p = \{p[1], p[2] \ldots p[n]\}$ all paths in ODG.
    For each path $p[i]$, $n = Node$, $\forall p[i] \in p$ Apply CPM method as discussed in section 2.3
    $t_i \leftarrow TC$ // generate test cases for all paths using CPM
    While $(n \neq N(p[i]))$
    $t_i = t_i + t$
    $C = N_x$
    End While
    $TS = TS + t_i$
    End for
```

Step 4: Analyze OCL based coverage criteria.

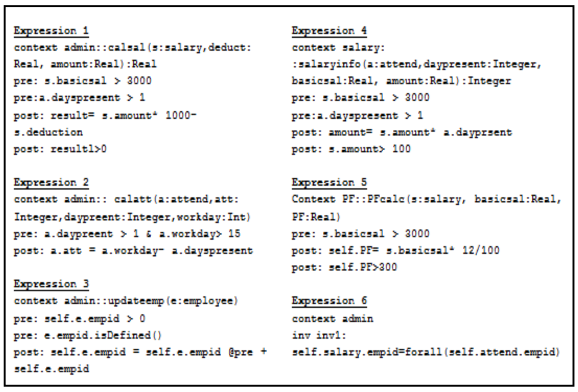
5. EXPERIMENTATION

This section briefs the implementation of the proposed algorithm. The algorithm is tested with various OCL constraints. The proposed algorithm is coded in Java. This section explains the OCL-based test case generation with CPM for simple PAYROLL System.

Table.1. Object Dependency Table for PAYROLL System

Component ID	Method with Object Name	Object Dependency based on DCC	Edge ID
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

Step 1: OCL expressions of the PAYROLL system are derived from its requirement specification document and saved as payroll.txt as shown in Fig.2.



```
Expression 1
context admin::calsal(s:salary,deduct:
Real, amount:Real):Real
pre: s.basicsal > 3000
pre:a.dayspresent > 1
post: result= s.amount* 1000-
s.deduction
post: result1>0

Expression 2
context admin:: calatt(a:attend,att:
Integer,daypreent:Integer,workday:Int)
pre: a.daypreent > 1 & a.workday> 15
post: a.att = a.workday- a.dayspresent

Expression 3
context admin::updateemp(e:employee)
pre: self.e.empid > 0
pre: e.empid.isDefined()
post: self.e.empid = self.e.empid @pre +
self.e.empid

Expression 4
context salary:
:salaryinfo(a:attend,daypresent:Integer,
basicsal:Real, amount:Real):Integer
pre: s.basicsal > 3000
pre:a.dayspresent > 1
post: amount= s.amount* a.dayprsent
post: s.amount> 100

Expression 5
Context PF::PFcalc(s:salary, basicsal:Real,
PF:Real)
pre: s.basicsal > 3000
post: self.PF= s.basicsal* 12/100
post: self.PF>300

Expression 6
context admin
inv inv1:
self.salary.empid=forall(self.attend.empid)
```

Fig.2. OCL expressions of the PAYROLL system

Step 2: Test path Generation - This section includes two sub modules, namely dependency table generation and graph construction.

Step 2.1: The dependency between each component of the PAYROLL system is examined by deriving its DCC metric value from payroll.txt. Based on the object dependency, the edges between the components are identified. Then, ODT is constructed for the PAYROLL system, which represented in Table.1.

Step 2.2: Based on the object dependency table values the dependency graph is constructed in which node represents the components name and edges connects the dependency components. Fig.3 represents the ODG for the PAYROLL system.

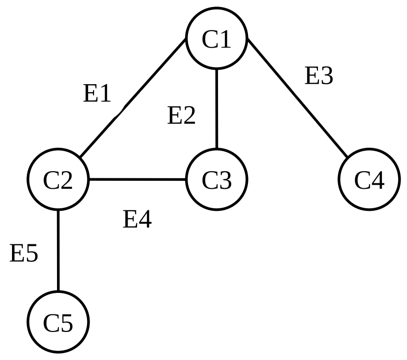


Fig.3. Object Dependency Graph for the PAYROLL System

Step 2.3: Traverse the graph through the depth first manner to generate test paths. There are three paths generated for the PAYROLL system. Here, TRP represents the test path requirement. The generated test paths cover all the nodes and edges of its ODG.

TPR = {C1, C2, C3, C4, C5}

Test path 1: C1→ C2→C5

Test Path 2: C1→ C3 → C2→C5

Test Path 3: C1→ C4

Step 3: As discussed in section 2.3, CPM is used to generate specification based test case. The steps are given below.

Table.2. Parameters, Pre and Post Conditions for the PAYROLL System

Functional Unit with Object Name	Parameters	Pre-conditions	Post-conditions
C1: calsal	s:salary,deduct: Real, amount:Real	s.basicsal > 3000 a.dayspresent > 1	s.basicsal* a.dayspresent* 1000- s.deduction result>0
C1: calatt	a:attend, att:Integer, daypreent:Integer, workday:Integer	pre: a.daypreent > 1 pre:a.workday> 15	a.att = a.workday- a.dayspresent
C1: updateemp	e:employee	pre: self.e.empid > 0 pre: e.empid.isDefined()	self.e.empid = self.e.empid @pre + self.e.empid
C2:salaryinfo	a:attend, daypresent:Integer, basicsal:Real, amount:Real	pre: s.basicsal > 3000 pre:a.dayspresent > 15	post: amount= s.amount* a.dayprsent post: s.amount>100
C5: PFcalc	PFcalc(s:salary, basicsal:Real, PF:Real)	pre: s.basicsal > 3000	post: self.PF= s.basicsal* 20/100 post: self.PF>300

Step 3.1: The functional units or methods of the PAYROLL system are extracted from payroll.txt, which include 'calsal', 'calatt', 'updateemp',' salaryinfo', 'PFcalc'.

Step 3.2: The various entities, include parameters, pre and post conditions of the PAYROLL system are extracted from payroll.txt, which are listed in Table.2.

Step 3.3: Categories defines the major characteristics of each parameter and environment condition hence, it affects the execution behavior of the system. Moreover, it is analogous to equivalence classes and it is a subset of parameter values. The Table.3 presents the categories for the method 'salaryinfo' of the PAYROLL system.

Table.3. Categories for the functional unit salaryinfo of the PAYROLL System

Functional Unit	Parameters, Precondition, Post condition	Categories
C2:salaryinfo	a:attend daypresent:Integer basicsal:Real amount:Real	a:attend Daypresent< =0 Daypresent>=0 Basicsal >=3000 Basicsal<=3000 amount:>=0 amount: <=0
Precondition	s.basicsal > 1	s.basicsal > 3000 s.basicsal = 3000 s.basicsal < 3000

Post condition	amount= s.amount* a.dayprsent
	s.amount>0
		amount= s.amount* a.dayprsent
		amount= s.amount+ a.dayprsent
		amount= s.amount- a.dayprsent
		amount= s.amount*- a.dayprsent
		s.amount>300
		s.amount<300
		s.amount=300

Step 3.4: A choice for each category is specified by providing all different kinds of values that are possible for it. It is similar to boundary conditions. Furthermore, it is a specific test value, which includes both valid and invalid values for each category. The Table.4 depicts the choices for the method 'salaryinfo' of the PAYROLL system. Due to the space constraint the number of choices for each entity is limited in this paper.

Sep 3.5: The constraints among choices are identified according to method invariants and environment conditions of the system. The Table.5 represents the constraints which are applicable to various choices of the method ‘salaryinfo’ of the PARYROLL system.

Step 3.6: The cross-product of all choices defines the test cases. Generate test cases by selecting a single element from each choice and stored in a separate table. The Table.6 depicts test cases and their expected results for the method ‘salaryinfo’ of the PAYROLL system.

Step 4: Coverage Analysis - There are four coverage criteria are used in the proposed approach to test the adequacy criteria. As discussed in the section 2.4 the coverage values are calculated for all components of the PAYROLL system which is represented in Table.7.

Table.4. Choices for the functional unit salaryinfo of the PAYROLL System

Functional Unit	Parameters, Precondition, Post condition	Categories	Choices
C2:salaryinfo	a:attend daypresent:Integer basicsal:Real amount:Real	a:attend Daypresent< =0 Daypresent>=0 Basicsal >=3000 Basicsal<=3000 amount:>=0 amount: <=0	a:attend Daypresent< 0, Daypresent< =0 Daypresent>0, Daypresent=0 Basicsal >0, Basicsal =0 Basicsal<0, Basicsal=0 amount:>0, amount:=0 amount: <0, amount: =0
Precondition	s.basicsal > 1	s.basicsal > 3000 s.basicsal < 3000 s.basicsal = 3000	s.basicsal > 3000 s.basicsal <3000 s.basicsal=3000

Post condition	amount= a.dayprsent
	s.amount>0 s.amount*
		amount= s.amount* a.dayprsent
		amount= s.amount+ a.dayprsent
		amount= s.amount- a.dayprsent
		amount= s.amount*-a.dayprsent
		s.amount>0
		s.amount<0
		s.amount=0
			amount= s.amount* a.dayprsent
			amount= -1
			amount= -1
			amount= -1
			s.amount>0
			s.amount=-1
			s.amount=0

Table.5. Constraints for the functional unit salaryinfo of the PAYROLL System

Functional Unit	Choices	Constraints
C2:salaryinfo	a:attend Daypresent< 0, Daypresent< =0 Daypresent>0, Daypresent=0 Basicsal >0, Basicsal =0 Basicsal<0, Basicsal=0 amount:>0, amount:=0 amount: <0, amount: =0	a.attend dayspresent >0 basicsal >0 amount>0
Precondition	s.basicsal > 3000 s.basicsal <3000 s.basicsal=3000	Basicsal >3000

Post condition	amount= s.amount* a.dayprsent
	amount<100
	amount= 0
	amount= -1
	s.amount>0
	s.amount=-1
	s.amount=0
		let sal:Real=0 in
		if self.attendance.noofpre=0
		then
		sal=0
		else
		basicsal=sal
		endif
		totalsal <=1 implies
		self.attendance.noofpre <=0 or
		self.attendance.noofhol >=30 or
		self.attendance.workday <=0

Table.6. Constraints for the functional unit salaryinfo of the PAYROLL System

Test Case	Functional Parameter / Choices				Expected Result
1	a:attend	Daypresent> 0	Basicsal >3000	amount:>0	Valid
2	a:attend	Daypresent> 0	Basicsal <3000	amount:>0	Invalid
3	a:attend	Daypresent> 0	Basicsal >3000	amount:<0	Invalid
4	a:attend	Daypresent< 0	Basicsal >3000	amount:>0	Invalid
5	a:attend	Daypresent< 0	Basicsal >3000	amount:<0	Invalid
6	a:attend	Daypresent> 0	Basicsal >3000	amount:=0	Invalid
7	a:attend	Daypresent> 0	Basicsal <3000	amount:>0	Invalid
8	a:attend	Daypresent= 0	Basicsal >3000	amount:>0	Invalid
9	a:attend	Daypresent= 0	Basicsal >3000	amount:<0	Invalid
	Pre and Post conditions				
10	s.basicsal > 3000	amount= s.amount* a.dayprsent		s.amount>0	Valid
11	s.basicsal < 3000	amount= s.amount* a.dayprsent		s.amount>0	Invalid
12	s.basicsal = 3000	amount= s.amount* a.dayprsent		s.amount>0	Invalid
13	s.basicsal > 3000	amount= s.amount+ a.dayprsent		s.amount>0	Invalid
14	s.basicsal > 3000	amount= s.amount*- a.dayprsent		s.amount>0	Invalid
15	…	…		…	…

6. COMPARATIVE STUDY

Many of the existing studies reported in the related work showed that there are different approaches [3, 4, 6 and 7] used to generate test cases from Z specification. Our approach uses OCL formal specification and CPM to facilitate test case generation from OCL.

Specification related test generation methods require formal specifications with specific interpretations to generate test data or an additional formalism such as higher order logic transformation, XML schema definition etc. On the other hand, many of the specification-based testing works [3 and 4] have reported the manual methods for test data generation.

A.D. Brucker et al.[7] have proposed a method to generate test data based on higher order representation of OCL specification. In their approach, the OCL specification of the system is first transformed into higher order logic. Then, object graph is constructed. However, their approach does not take into account the coverage criteria of the program.

In our approach, we use exactly the same OCL specification derived for precise requirement or model specification, without requiring any additional transformation or effort specifically meant for testing purposes. The study of A.D. Brucker et al.

focused only on the automatic test cases generation. Tests path and coverage have not been analyzed properly.

The limitation of the prior work [7] is, it focuses on OCL pre, post conditions and invariants and it ignores method parameters for test data generation. Therefore, this test case generation technique could not detect all abstract system level bugs. A.D. Brucker et al. have generated the object graph based on the concept of alias closure.

However, it involves complex processes and mathematical operations. In our proposed work, the test path is generated using DCC metric which can be derived easily from OCL specification of the SUT.

The work of A.D. Brucker et al. is derives test data, but it is often complex by transforming OCL specification into higher order logic. It is inferred that the derivation of higher order logic from OCL specification is the complex operation. As a result of this, test efforts are wasted which increases the testing cost.

Table.7. Constraints for the functional unit salaryinfo of the PAYROLL System

Components	**Coverage Criteria**			
	Statement	**Parameter**	**Pre-Post condition**	**Path**
C1: admin	80%	92%	75%	100%
C2: salary	90%	97%	88%	85%
C3:attendance	94%	92%	77%	89%
C4: employee	77%	79%	92%	94%
C5:PF	87%	88%	79%	77%

Unlike A.D. Brucker et al. work, our test case generation approach detects statement, condition and parameter wise faults in the OCL specification of the SUT. There is no redundant test data have been generated by our approach. The automation of our proposed approach is very simple.

Moreover, the corrective measure at the early phases of software development is more effective than at the later stages of software development. Our proposed approach is focused on the specification time testing, which is more effective than design time testing. We compared our work with the existing work on specification-based test case generation [7] which is depicted in Table.8.

Table.8. Comparison of our approach with A. D. Brucker et al. work

Sl. No	**Comparison criterion**	**A.D. Brucker et al. approach**	**Our approach**
1	Type of testing	Model based	Specification based
2	Methodology Adopted	Theorem-prover- based	Category Partitioning Method
3	Parameter Included	No	Yes
4	Intermediate representation	Higher order logic	Categories and Choices
5	Graph used	Object Graph	Object dependency Graph
6	Path Generation Technique	Alias Closure	DCC metric
7	Coverage analysed	No	Yes
8	Automation	Difficult	Simple
9	Effectiveness	In appropriate for complex system	Applicable to all type of systems and systematic

7. CONCLUSION

In this research work, we have adopted the functional test case generation technique using OCL and CPM. The proposed approach can generate high efficient test cases at the specification level. The major merits of the framework are:

The dependency relations between components are realized using DCC metrics value. Dependency graph serves to construct test paths. Test cases are generated directly from formal specification of the system. The proposed work adopts the systematic way to control the maximum number of generated test cases. It serves to detect and correct faults due to incorrect specification.

We have applied our approach to many real-time applications and observed the effectiveness of specification based testing using its categories and choices. The proposed approach shall generate test cases efficiently and thus improves overall software quality.

As a future work, it has been proposed to adopt search based techniques, so as to generate and prioritize test cases based on OCL specification of the system.

ACKNOWLEDGMENT

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

REFERENCES

[1] Joseph Gil, John Howse and Stuart Kent, “Constraint Diagrams. A Step beyond UML”, *Technology of Object-Oriented Languages and Systems*, pp 1-10, 1999.

[2] Philippe Collet and Roger Rousseau, “Towards Efficient Support for Executing the Object Constraint Language”, *Proceedings of Technology of Object-Oriented Languages and Systems*, pp. 399-408, 1999.

[3] Ben Potter Jane Sinclair and Jane Sinclair, "*An Introduction to Formal Specification and Z*", 2nd Edition. Prentice Hall, 1996.

[4] Percy Antonio Pari Salas and Bernhard K. Aichernig, "Automatic Test Data Generation for OCL: A Mutation Approach", *Proceedings of 5th International Conference on Quality Software*, pp. 64-71, 2005.

[5] Li Bao-Lin, Li Zhi-shu, Li. Qing, and Chen Yan Hong, "Test Case Automate Generation from UML Sequence Diagram and OCL expression", *International Conference on Computational Intelligence and Security*, pp. 1048-1052, 2007.

[6] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation", *Proceedings of 11th International Conference on Quality Software*, pp. 41-50, 2011.

[7] Achim D. Brucker, Matthias P. Krieger, Delphine Longuet and Burkhart Wolff, "A Specification-Based Test Case Generation Method for UML/ OCL", *Proceedings of International Conference on Models in Software Engineering*, pp. 334-348, 2011.

[8] Thomas J Ostrand and Marc J Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, Vol. 31, No. 6, pp. 676-686, 1988.

[9] Jagadish Bansiya and Carl Davis, "Automated Metrics and Object-Oriented Development", *Dr. Dobb's Journal*, pp. 42-48, 1997.

[10] N. Amla and P. Ammann, "Using Z Specifications in Category Partition Testing", *Proceeding of the Seventh Annual Conference on Systems Integrity, Software Safety and Process Security: Building the System Right*, pp. 3-10, 1992.

[11] Paul Ammann and Jeff Offutt, "Using Formal Methods to Derive Test Frames in Category-Partition Testing", *Proceedings of 9th Annual Conference on Computer Assurance*, pp. 69-80, 1994.

[12] Jeff Offutt and Alisa Irvine, "Testing Object-Oriented Software using the Category-Partition Method", *Proceedings of 17th International Conferences on Technology of OO Languages and Systems*, pp. 293-303, 1995.

[13] Matthias Grochtmann and Klaus Grimm, "Classification trees for partition testing", *Software Testing, Verification and Reliability*, Vol. 3, No. 2, pp. 63-82, 1993.

[14] T.Y. Chen, Pak Lok Poon and T.H. Tse, "A Choice Relation Framework for Supporting Category Partition Test Case Generation", *IEEE Transactions on Software Engineering*, Vol. 29, No. 7, pp. 577-593, 2003.

[15] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini, "Automatic test data generation for XML Schema-Based Partition Testing", *Proceedings of 2nd International Workshop on Automation of Software Test*, 2007.