Generating Object-Oriented and Aspect-Oriented Sequence based Test Cases with Optimum Code Coverage

Annasaro Vijendran, PhD.
Director, Dept of Computer Applications
SNR Sons College, Nava India
Coimbatore, Tamilnadu

N.R.Suganya
Research Scholar, Karpagam University
Eachanari, Coimbatore
Tamilnadu

ABSTRACT
Code coverage analysis holds a very important role in software testing procedure. By the test case runs it provides developers by means to quantify of how well their source code is being exercised. By detecting errors/bugs in the code it estimates the efficiency of the test. We must implement a systematic way and support the theoretical bases for testing the programs with the purpose of performing effective software testing and coverage analysis. In our research we use the crossword application where we automatically make test cases and systematically discover the impact of context, as captured by criterion functions which we described in our source code. Our studying demonstrates that by increasing the event combinations tested and by organizing the comparative positions of events defined by the new criteria, we can become aware of a large number of faults that were undetectable by earlier techniques. In this paper we are implementing the event based test case generation by the use of object oriented and aspect oriented event based test case generation. Before this we have used the GUI based test case generation as an existing work. The experimental result shows that our proposed work test case generation process providing better code coverage range when compared with the existing work.

Keywords
Code Coverage, Aspect oriented testing, Object oriented testing, GUI testing, automated testing, test case generation, Testing Process.

1. INTRODUCTION
Software testing is a vital phase of software development. It gives a method to set up assurance in the trustworthiness of software. Software testing is an extremely manual labor-demanding job, which uses the majority reserve in test data creation (TDC). If the development of testing is computerized this expenditure might be diminished. TDC is defined as follows: for a given program source code, discover a program input on which this source code is executed [2]. TDC comprises random, path-oriented, and goal-oriented test data generation [3]. Random TDC is obtained as the minimum acceptable orientation point to additional assessment techniques. Path-oriented TDC take account of constraint-base TDC [4], dynamic domain reduction TDC [5], and an improved method introduced in reference [6]. Goal-oriented

TDC includes chaining method for TDC [2] and assertion-oriented approach [7]. We see four major aspects of software testing software development which should be addressed in automation systems development: These aspects of testing include (a) test sketching, association, and observation, (b) test case generation, (c) test case completion and execution, and (d) test statement on various levels. The earlier studies focused on technical aspects of software testing more willingly than on organizational aspects. Acceptance and organizations tests focal point is on customer requirements and business cases from end user perspective (top level).

Integration tests focus on the architectural design of the system, individual components, and the interaction between components, and unit tests include testing of individual components [12]. In this paper we present a framework for automated testing of industrial control automation applications and discuss individual aspects of a basic testing process and how these aspects address various systems levels. These techniques can automatically generate many test cases, whose effectiveness depends on the completeness of the initial model. When the initial representation is obtained by stimulating the application under test by means of a simple sampling approach that makes use of a subset of GUI actions to find the way and examine the casements, the derived model is limited and incomplete, and the generated test cases unavoidably fail to notice that many connections and windows not determined in the initial phase. In proposed work we are implementing the object oriented testing and aspect oriented testing to improve the code coverage estimation from the test case generation. When dealing with an object-oriented (OO) vital language, representative execution turns out to be challenging as, between other things, it must be capable to backtrack, composite heap-allocated data structures should be created at some stage in the (Test Case Generation) TCG process and features like inheritance, virtual incantations and exclusions have to be taken into account.

Software testing is a recurrent problem; as a result it scores limited concentration. In addition to testing confronts is aspect-oriented model, which has a dichotomy of core and crosscutting concerns. Since emergent behavior of the aspects during their interaction with objects, and inter dependencies not only incurring challenges for testing, but also alludes to creation of inventive testing techniques. Several faults are introduced by aspects. In this paper, review has been done on all the existing testing techniques including static verification techniques intended for aspect-oriented programs. These factors request us to work out an inclusive testing framework to meet up obliviousness of damaging aspects. Various approaches are lacking in automation or either can't deal with testing of large programs. After the implementation of these two testing we are focusing on the coverage or adequacy which is a software testing metric that is used to assess the

relevancy or the efficiency of the engender test suite accumulatively. There are more than a few types of concerns in the application that can be used to estimate coverage. Some of those coverage criteria are code supported such as: statement, decisions, and paths coverage. At the side of code criteria, coverage can be calculated based on obligation or design criteria.

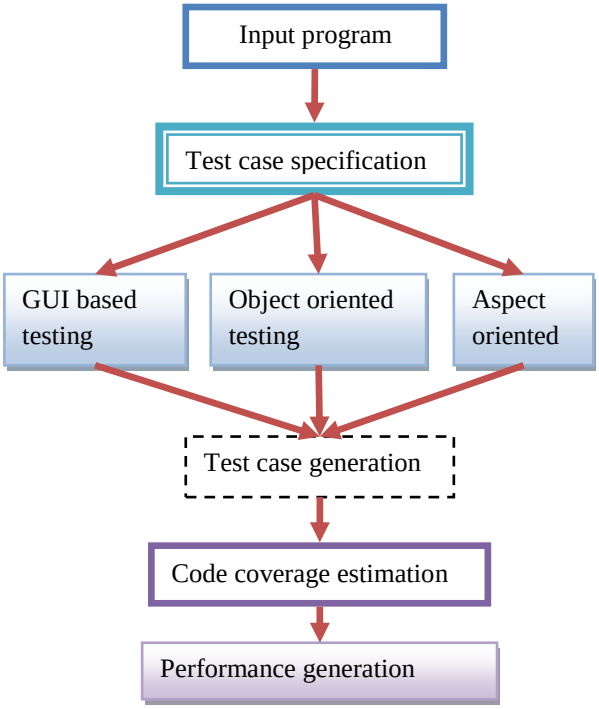

Fig 1: Block diagram of detecting domain error

The main contribution of works is as follows:

1. Creating the test cases based on the object oriented paradigm
2. Aspect oriented test case generation for our application
3. Code coverage estimation

The remainder of the paper is as follows. In section 2 the GUI testing is described. The object oriented testing is explained with corresponding algorithm in section 3. In section 4 the aspect oriented testing is explained. In section 5 the code coverage estimation is described. In section 6 results and discussion following with this conclusion is described in section 5.

In this research we are developing a cross word application program which can be the input to the testing process. Where the GUI testing, Object oriented testing, Aspect oriented testing is done. In this the true positive results and false positive results are validated to find the code coverage.

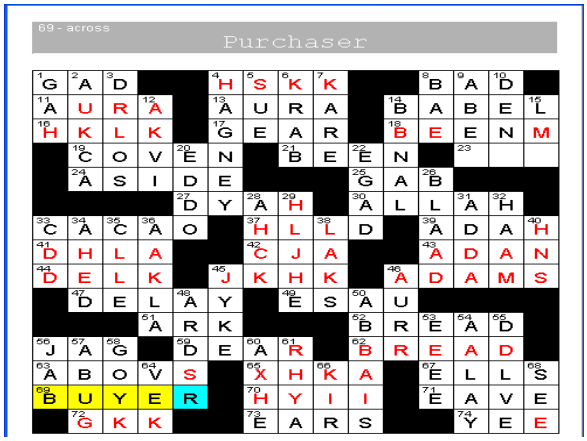

Fig 2: The cross word application

2. GUI MODEL BASED TESTING

Test coverage is a universal term that is used to assess one or more concerns in the application and analyze the percentage that the test cases in a test suite are covering. Illustrations of test coverage types used are: statement, decisions, paths, etc. As we produce a GUI model in existing work that comprises the GUI modules, their characteristics and relations, several coverage concerns can be evaluated from this model. This includes: GUI paths, nodes, edges and components coverage. Bulky remains of research on software testing for GUIs are present and many GUI testing techniques have been proposed; some have been implemented as tools and accepted by practitioners. All of these techniques automate a quantity of aspect(s) of GUI testing, including model creation (for model-based testing), test case generation, test vision creation, test execution, and regression testing. Even though the character and type of test cases possibly will differ with diverse techniques, all of them discover the GUI's state space via series of GUI events. Semi-automated unit testing tools, such as JFCUnit, Abbot, Pounder, and Jemmy Module, are used to manually create unit GUI test cases, which are then automatically executed. Declarations are inserted in the test cases to decide whether the classes/methods in the unit under test function are running correctly. More advanced tools, called capture/replay tools, "capture" a user session as a test case, which can later be "replayed" automatically on the GUI.

Again, test creation is physical and the tools make possible only the execution of test cases. The fraction of the GUI state space explored by these test cases depends largely on the experience and knowledge of the testers and the excellence of the user assemblies. Model-based techniques have been used to automate certain aspects of GUI testing. For instance, manually created state machine models have been used to generate test cases. The nature and fault-detection effectiveness of generated test cases depend largely on the definition of "GUI states." Other work on GUI testing has focused on graph models to minimize manual work. The most successful graph models that have been used for GUI test-case generation include EFGs and EIGs. The nodes in these graphs represent GUI events; edges represent different types of relationships between pairs of events.

An EFG (Event Flow Graph) models all achievable event sequences that may be executed on a GUI. It is a directed graph that contains nodes and edges that represent a relationship between events one for each event in the GUI. An

edge from node n_x to node n_y means that the event represented by n_y may be executed instantly after the event represented by n_x. This association is described as follows. Make a note of that a state machine model that is correspondent to this graph can also be bring together and the state would detain the potential events that can be performed on the GUI at any instant; transitions origin state changes on every occasion the amount and type of available events change. The EFG is represented by two sets: 1) a set of nodes N on behalf of events in the GUI and 2) a set E of prearranged pairs (e_x, e_y), everywhere $\{e_x, e_y\} \subseteq$ N, instead of the directed edges in the EFG; $\{e_x, e_y\} \in$ E iff e_y follows e_x. A main property of a GUI's EFG is that it can be created semi-automatically using a reverse engineering technique called GUI Ripping. A GUI Ripper automatically passes through a GUI under test and removes the hierarchical configurations of the GUI and events that could be executed on the GUI. The result of this process is the EFG. EIG (Event-Interaction Graph) nodes, conversely, do not stand for events to open or close menus, or open windows. The result is a more packed together and for this reason more efficient, GUI model. An EFG can be automatically transformed into an EIG by using graph rewriting rules. While doing this GUI based testing it can be efficient testing model but also having the disadvantages.

2.1 Algorithm for GUI Event Based Test Case Generation

Input: Input sequence from the user

Output: Test case generation

STEP 1: Let we denote the GUI state S which has collection of objects $\{o_1, o_2, \ldots . o_n\}$

STEP 2: Each object represented with their type and property (with their values) which is represented as $(type_i, P_i)$

STEP 3: Write abstract state AS from GUI state S using an abstraction function as denoted below:

$$proj(S) = proj, AS$$

STEP 4: The function $proj$ is a projection operator which is applied to every objects in the state S, where

$$proj(S) = \{proj(o_1), proj(o_2), \ldots . proj(o_n)\}$$

STEP 5: Given an object o_i, $proj$ extracts a subset of its properties, that is $proj(o_i) = \left(type_i^{'}, P_i^{'}\right)$ where $\left|P_i^{'}\right| \leq |P_i|$

STEP 6: For each type of object, we defined the $proj$ function to extract the properties that carry semantically useful information.

STEP 7: The abstract state returned by the abstraction process is the state representation that is incorporated in the behavioral model for each event process.

STEP 8: By extracting an abstract representation of the GUI state, the developer checks if the same objects occur in multiple GUI states S, possibly with some modified properties.

STEP 9: The test case developer identifies occurrences of the same object in multiple states by comparing the object in a GUI with the objects representation by the behavioral model.

STEP 10: To detect multiple occurrences of the same object, we defined a function that generates attributes from objects. An attribute is a subset of the widget properties that are expected to be both representative and invariant for the given object.

STEP 11: Using these attributes, we can formally define a concept of identity. Given two objects o_1, o_2 if $att\ (o_1) = att\ (o_2)$ then we can say that $o_1 = a = o_2$.

STEP 12: Like, the attribute of a button includes the type of the objects, the position of the button in the GUI hierarchy, and the label visualized on the button are written for each event e_i.

STEP 13: Based on these definitions we can define the following restriction operator between abstract states: Given $AS = \{o_1, o_2, \ldots . o_n\}$ and $AS^{'} = \{o_1^{'}, o_2^{'}, \ldots . o_n^{'}\}$, abstract states, $AS \backslash_t AS^{'} = \left\{o_i \middle| o_i \in AS \wedge \nexists_{o_k} \in AS^{'}s.t\ o_i =_t o_k\right\}$

STEP 14: Output of the test case generation

While doing this GUI based testing it can be efficient testing model but also having the disadvantages. They are all described as follows:

2.2 Challenges of Using GUI Model Based Testing

Applying model based testing techniques for testing user interfaces may face some challenges.

1. The first challenge is dealing with the state explosion problem where there will be a large number of possible GUI states in even simple applications.

2. One of the majority limitation of GUI models is that they are produced manually which makes the procedure burdensome and do not rationalize its practice on actual testing activities where resources are already fixed and limited.

3. A GUI design model may not be without difficulty or automatically converted to an implementation model. In this follow a line of investigation we didn't employ model transformation to transfer a design model into an achievement one.

4. Generating a GUI model manually is time consuming and may substantiate omitting such stage trading off this effort with its possible advantages.

This problem can be avoided through automation. Nevertheless, building such model automatically is not a minor task. The suggestion that this paper is built on is that using model based formal verifications automatically may improve software testing in terms of time (i.e. automation) and coverage. These challenges are avoided and the domain error/bugs are identified with the implementation of object oriented and aspect oriented testing.

3. OBJECT ORIENTED TESTING

Object-orientation has hurriedly become traditional as the preferred paradigm for large scale system design. The reasons for this are first, classes provide an excellent structuring mechanism. They agree to a system to be separated into well

defined components which may then be implemented separately. Classes support information-hiding is considered to be second. A class can export a purely procedural edge and the internal structure of data may be concealed. This permits the configuration to be changed without disturbing users of the class, thus simplifying preservation. Third, object-orientation gives confidence and holds up software reprocess. This may be accomplished moreover through the simple reuse of a class in a library, or by means of inheritance, whereby a new class may be fashioned as a conservatory of an obtainable one.

In both cases the result is a decrease in the quantity of software which must be written and, as a result, a development in the consistency of the ensuing system in view of the fact that in the past tested programs may be consumed. If we are to take advantage of on the latent compensation of object-orientation then it is imperative that the object-oriented approach is accepted and maintained throughout the software development process. The design phase is now practically well understood (although disreputably difficult) and there are various tools to support with the process. Likewise there are tools to support the accomplishment phase. Library browsers may be used to aid with locating existing classes which may be reused in the project and tools provide support for editing, compilation, etc.

Though, testing is often ignored by the designers of software tools and the programmer is absent to his/her own resources. Some may disagree that testing should not be difficult (or even necessary at all) if a proper design and implementation process has taken place. We all recognize this not to be true and we must topic new software to thorough testing previous to it can be used in a production situation. The structuring of the system as a set of autonomous classes obliges that each of these must be tested and there may be a large number of them. In addition, information-hiding, which gives confidence designers of classes to have with the sole purpose technical interfaces, makes it difficult to decide whether the class is working correctly, because the state of inside data may not be easy to get to via the interface. The consequence is that the programmer must efficiently expand a test program for each class. Each such test program must create an instance of the class being tested and include calls to each of the techniques sustained by the class. The test program will need to prompt the user for the parameters for these method calls so that a variety of combinations can be worked out. The test program must also demonstrate the results of each method call. The result is that the test program may well be more complex and larger than the class being tested. Once we have written such a test program, we at a standstill may not be able to determine whether the internal data of the class being tested is correct because of the inability to access all of the internal data via the procedural interface.

3.1 Object Oriented Event Based Test Case Generation

Input: A state transition graph Output: Test case set

Procedure Transition graph Generation
$let\ V = \{t_i, t_j\}$, A:=NULL
For each $t \in T_{legal}$ do
$V = V \cup \{t\}$
For each $S \in S_{simple}$ do

For each $t \in in(s)$ do
For each $t' \in outt(s)T_{faulty}$ do
$A = A \cup \{t, t'\}$
End for
End fro
For each $t \in out(initial(root()))T_{faulty}$ do
$A = A \cup \{t_i, t\}$,
End for
For each $S \in S_f$ do
For each $t \in in(s)$ do
$A = A \cup \{t, t_j\}$
End for
End for
call transition graph generation (V_t, E_d)
$TS = \varphi$
 	
For k=2 to n do
For each $v \epsilon V$ do
If$(t_i, V) \in E_d$
$TS = TS \cup V$
$Ed = Ed \cup (t_i, t_j)$
Apply the graphical traveling salesman problem
Add them to set TS
Endif
Endfor
Enddo
Enddo
Enddo
End
Endfor

4. ASPECT ORIENTED TESTING

The main idea of aspect oriented model is that aspects facilitate the modular demonstration of crosscut word formation. An apprehension is a dimension in which a word formation is prepared, and is crosscut word generation if it cannot be realized in traditional object-oriented designs apart from with scattered and tangled code. By scattered we mean not localized in a module but sectioned across a system. By twisted we mean interacted with code for other concerns. The below given is the test case generated for the sample cross word application fig 2.

{ System.out.println("Check NSFE");}
System.out.println("A constructor call to Crosswordd is about to occur");
System.out.println("A method execution in Crosswordd is about to occur");
System.out.println("A constructor execution in Crosswordd is about to occur");
System.out.println("Crosswordd is about to undergo instance initialization");
System.out.println("Crosswordd is about to undergo class initialization");
System.out.println("A exception of type Crosswordd is about to be handled");
System.out.println("A constructor call to Crosswordd just occurred");
System.out.println("A method execution in Crosswordd just occurred");
System.out.println("A constructor execution in Crosswordd just occurred");

```
System.out.println("Crosswordd has just undergone instance initialization");
System.out.println("Crosswordd has just undergone class initialization");
System.out.println("A exception of type Crosswordd has just been handled");
}
```

An aspect performs behavioral modification to program execution event exposed by the language definition. The aspects are defined in above code. It is designed in an extraordinary fashion to detect faults; therefore, harmful aspects are easy to witness. We provide the crossword which is an application regression testing because this testing strategy focuses on to locate undesirable changes which also referred as errors made by aspects in the new application. To examine the contact of aspects on the appropriateness of center concerns we can take on this regression testing framework. Generally, regression testing is applied on application and modules. It is performed to correct faults, steps up in functionalities.

4.1 Aspect Oriented Event Based Test Case Generation

STEP 1: Get Information about the class's information; write the aspects and their categories determined
STEP 2: Get the testing criterion
STEP 3: LOAD step 1 Information in CONSTRUCTOR
STEP 4: Regression Fault Information Updating if already exists
STEP 5: Write aspect criterion
STEP 6: To trace existing Test Cases using Comparator for Reuse with including existing test cases applied to aspects already
STEP 7: Create New Test Cases if criterion/ testing environment don't satisfy also generate Test Data along with that.
STEP 8: Starting testing process for verification by tester
STEP 9: Complete Setup for testing environment
STEP 10: while (testing criterion not true)
STEP 11: test the aspect oriented programming function
STEP 12: for(i=0; i< Test_Cases; ++i)
STEP 13: if (Test (i) = true)
STEP 14: save the result
STEP 15: else interpret the test case result
STEP 16: end for
STEP 17: end while
STEP 18: End of Testing Activity

5. CODE COVERAGE ANALYSIS

Comprehensive testing refers to exercising all possible inputs of a program in order to prove the absence of faults. Unfortunately, in the typical case of vast or infinite input spaces, exhaustive testing is infeasible, and any sub-set of an exhaustive test set can only show the presence of bugs not their absence. Even though exhaustive testing is not always possible, thorough testing can provide a basis for sufficient confidence in program behavior. A test adequacy criterion is a predicate defining what properties of a program must be exercised to constitute a thorough test. Adequacy criteria help determine when testing may cease. Code coverage analysis is a technique to determine whether a set of test cases satisfy an adequacy criterion. It helps find areas of a program not sufficiently exercised by a set of test cases, gives a measure of the quality of a test set, and can determine whether a test set meets coverage requirements. In general, adequacy is assessed against selected parts of a code base, and those parts are scattered across the program.

Measuring coverage requires that the source code be instrumented at points of interest. Something like a break point in debugging has to be inserted at each point of interest. To instrument code manually is a burden on the tester. Code coverage tools are meant to help remove this burden by automating the task of code coverage analysis. Coverage tools are designed with specific adequacy criteria in mind. For example, a tool might allow statement coverage, path coverage, condition coverage, segment coverage etc. These tools, however, limit the flexibility of code coverage. The limited set of code coverage criteria is all a tester can choose from, and the code against which adequacy is to be evaluated can be designated only by the limited means supported by the given tool. In practice, testers are often not interested in coverage of all program elements, but rather in selected elements. One might hope to choose merely non-library and non-environment-generated code, for instance. Likewise, a tester of one module possibly would like to include the code of the component or task that is mortal tested and perhaps closely associated code elsewhere in the system, but not more faintly related code.

In the proposed cross word application for testing, it needs to convey selective test sufficiency decisive factor in the background of module based systems. We necessitate identifying the types of testing information about a component that a constituent user wants for testing applications that use the module. For illustration, a developer might want to determine coverage of the parts of the component that they application uses. The component must be able to react to inputs provided by the application, and evidence the coverage provided by those inputs to do this. For an additional illustration, a component user may want to test only the integration of the component with the application. To do this we have to be able to recognize couplings between the application and the component of the segment users.

Further commonly, a tester strength objective to define test sufficiency criterion in terms of coverage of code that communicates to concerns that are lexically multiply diagonally the code base. In the previous approach, general in profitable tools, the coverage tool makes available a fixed set of adequacy criteria, with the possibility of manually selecting code elements against which sufficiency is to be assessed. This approach does not offer an adequate amount of flexibility. The concluding approach provides a language front end to the examination tool. The suppleness in on behalf of adequacy criteria depends on the self-expression of this front-end language.

5.1 Pseudo Code for Code Coverage

TD : containing n test cases
test_file: test case file which is going to be executed
N : total number of test runs
Tot_Cov: total coverage of the program
Req_ Cov: required coverage value of program
STEP 1: Generate the input java program and give as input.
STEP 2: Initialize set total code coverage array [] =0 also assign variable to be 0 (Tot_Cov to 0).
STEP 3: Initially set N=0.

STEP 4: Provide initial test cases to the test_file and measure it during run.
STEP 5: Add the test case from the TD to the test_file based on required coverage value (Req-Cov)
STEP 6: Run the test_file
STEP 7: If (total code coverage[k] >total code coverage [k-1])
{
N = N + 1 and
Tot_Cov = total code coverage[k].
if (Tot_Cov >Req_Cov)
{
Return Tot_Cov and N values.
}
else
{
Repeat from STEP 5
}}
else
{
Repeat from STEP 5
}}
STEP 8: Return Tot_Cov and N values.

6. RESULTS AND DISCUSSIONS

6.1 True Positive Comparison

In this graph we are comparing the existing system and proposed system based on two parameters. In X axis we are having the methods and in Y axis we are having the values. From the graph true positive calculation is measured and in our proposed work we can see the best result when compared with the existing work. The true positive is like the testing criteria are satisfied as we are expecting is a positive test when testing phase this to be referred as true positive results. The result of the graph is shown below as Figure 3:

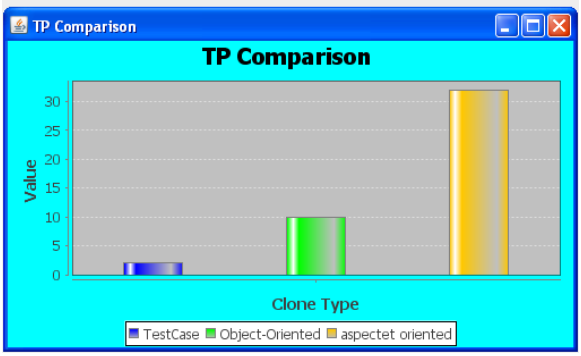

Fig 3: True Positive Comparison

6.2 False Positive Comparison

In this graph we are comparing the existing system and proposed system based on two parameters. In X axis we are having the methods and in Y axis we are having the values. From the graph true positive calculation is measured and in our proposed work we can see the best result when compared with the existing work. The false positive is like the testing criteria are satisfied as wrong when testing phase this to be referred as true positive results. The result of the graph is shown below as Figure 4:

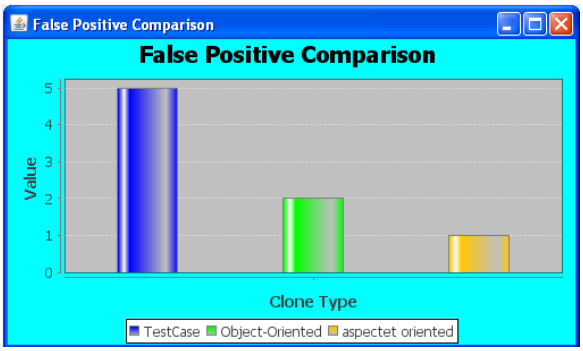

Fig 4: False Positive Comparison

6.3 Overall Performance

In this we are comparing the overall performance of the existing system and proposed system based on two parameters called test case which is to be X axis and the code covered which is to be the Y axis. When there is test cases increased the code coverage value of proposed aspect oriented approach is having the high amount of code coverage based on the test cases. From the graph we can say that the proposed work having high code coverage value. This is shown in Figure 5:

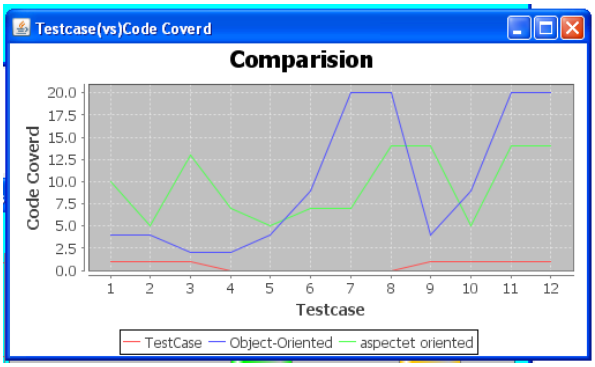

Fig 5: Overall performance

7. CONCLUSION

Exemplary research is in development for testing of aspect-oriented software. Significant concentration is rewarded to the challenging issues, which are situation by the new paradigm, but it is extremely significant that responsibility proven, high in cost, easier said than done to assume methods should not be visited in the perspective of aspect-oriented testing. It should also be on the program to take up the testing techniques for object-oriented systems and their bribes. We have presented a testing framework for aspect-oriented programs, which takes account of the analysis of aspects interaction with central part system. In this framework, we have recognized test cases, test data, reusability, automation aspects of the framework, and specifically addressing the mistakes, which are effectively removed during aspect-oriented programming oriented test case.

8. REFERENCES

[1] D. C. Kung, C. H. Liu, P. Hsia 2000 "An object-oriented Web test model for testing Web applications", in Proceedings of IEEE 24th Annual International

Computer Software and Applications Conference, Taipei, Taiwan pp. 537-542.

[2] R. Ferguson, B. Korel 1996 "The chaining approach for software test data generation" ACM Trans on Software Engineering and Methodology, pp. 63-86.

[3] Jon Edvardsson 1999 "A survey on automatic test data generation" Proceedings of the second conference on computer science and engineering pp. 21-28.

[4] R. A. DeMillo, A. J. Offut 1991 "Constraint-based automatic test data generation" IEEE Transactions on Software Engineering, pp. 900–910.

[5] A. J. Offutt, Z. Jin and J. Pan 1991 "The dynamic domain reduction procedure for test data generation. Software: Practice and Experience" pp. 900-910.

[6] J. H. Shan, Y. F. Gao, M. H. Liu etc. 2008 "A new approach to automated test data generation in mutation testing" Chinese journal of computer pp. 1025-1034.

[7] B. Korel, A. M. Al-Yami 1996 "Assertion-oriented automated test data generation" Proceedings of the 18th international conference on Software engineering, Berlin, Germany, pp. 71-80.

[8] Zhao, J. 2003 "Data-Flow-Based Unit Testing of Aspect-Oriented" Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03), In proceedings of IEEE.

[9] Rajan, H., Sullivan, K. 2005 "Generalizing AOP for Aspect-Oriented Testing" in conference 'ACM 1-58113-000-0/00/0004.

[10] Zhao, J., Rinard, M. "System Dependence Graph Construction for Aspect-Oriented Programs", Cambridge, USA.

[11] Hailpern, B., Santhanam, P. 2002 "Software debugging, testing, and verification" IBM SYSTEMS JOURNAL, VOL 41, NO 1.

[12] Mortensen, M., Alexander, R.T. 2004 "Adequate Testing of Aspect-Oriented Programs" Colorado State University, Fort Collins, Colorado, USA, Technical report CS 04-110.

[13] Xu, G., Yang, Z., Huang H., Chen, O., Chen, L., Xu, F.: JAOUT 2004 "Automated Generation of Aspect-Oriented Unit Test" Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), Also in proceedings of IEEE.

[14] Hughes, D., Greenwood, P. "Aspect Testing Framework" Computing Department, Lancaster University, UK.

[15] Augusto, O., Lemos, L., Maldonado, J.C., Masiero, P.C. "Data Flow Integration Testing Criteria for Aspect-Oriented Programs" Universidade de S˜ao Paulo, Av. do Trabalhador S˜ao-Carlense, 400, S˜ao Carlos, SP.

[16] Alexander, R.T., Bieman, J.M. 2004 "Towards the Systematic Testing of Aspect-Oriented Programs" Colorado State University, Department of Computer Science, Safety Systems Research Center, University of Bristol, Bristol, UK, Published by Elsevier Science B.V.