ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

An Overview on Test Case Generation using UML Models

Chayashree G

Assistant Professor, Department of ISE, GSSSIETW, Mysuru, India

ABSTRACT: one of the major quality criteria of a software system is how well it fulfills the needs of users or customers. One technique to verify and improve the grade of fulfillment is system testing. System test cases might be derived from the requirements of the system under test. The software testing immensely depends on three main phases: test case generation, test execution, and test evaluation. Test case generation is the core of any testing process; however, those generated test cases still require test data to be executed which makes the test data generation not less important than the test case generation. This kept the researchers during the past decade occupied with automating those processes which played a tremendous role in reducing the time and effort spent during the testing process.
This paper explores different approaches that had regarding the generation of test cases using UML models.

KEYWORDS: Model based Testing (MBT), Evolutionary testing, Genetic algorithms, metamorphic testing and Meta-heuristic search techniques.

I. INTRODUCTION

Software testing is becoming more complex day by day. This complexity enforces using techniques and methods to assure software quality. One of this methods is system testing. The main goal of system testing is to verify that requirements are successfully implemented into system under test. In order words, system testing assures that software system does what it is expected to do. Model based testing (MBT) refers to the type of testing process that focuses on deriving a test model using different types of formal ones, then converting this test model into a concrete set of test cases [1], [2], [3]. Those formal models have many different types, but all of them are generally categorized into three main categories: requirements models, usage models, and source code dependant models. The requirements models can be behavioral, interactional, or structural models according to the perspective by which the requirements are being looked at. The test cases derived from behavioral or interactional models are functional test cases and they have the same level of abstraction as the models creating them. These kinds of test cases differ from those derived using structural models. Other types of models can be used as well to extract test cases [3], [4], [5], [6]; The Unified Modeling Language (UML) models are considered one of the most highly ranked types being used [7]-[14].

Generated test cases require test data for their execution which makes the test data generation a building block activity in the overall test cases execution process. Test data can be derived from different UML models as well as other different types of models. Search based testing models are one of the most important models used; they include evolutionary models and Genetic algorithms [15]. In order to be able to claim that the generated test cases are better than others or even decide whether they are applicable or not; they must be first qualified for usage.

Coverage criteria are considered a set of metrics that are used to check the quality of test cases that are extracted from behavioral models [26], [27], [28]. This metrics' set contains many types of criteria and according to the UML model being used in generating the test cases a certain criterion or many criteria are selected rather than the others. Some examples of the coverage criteria are: The branch coverage criterion [29] and it is used with Control flow graphs. The full predicate and the condition coverage criteria in [30] are used to validate the test cases generated from state charts or communication diagrams. The all basic paths coverage criterion is another example mentioned in [31] and it is used with activity diagrams-based techniques.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

II. TEST CASE GENERATION TECHNIQUES

The more early test cases are generated, the more costs, time and effort can be saved when the actual testing time comes. Many researchers have recently given this field a great attention where test cases can be generated in the analysis and design phases using requirements-based models and sometimes other models. UML diagrams are the most common type of models used to represent the requirements-based models. They can be categorized into behavioral, interactional and structural diagrams [35].

- Behavioral diagrams are a type of diagrams that represent behavioral features of a system or business process. They include activity, state chart, and use case diagrams as well as the four interaction diagrams (communication, interaction, sequence and timing).
- Interactional diagrams are a subset of behavioral diagrams which accentuate object interactions. They include communication, interaction overview, sequence, and timing diagrams.
- Structural diagrams are a type of diagrams that emphasize the elements of a specification which are irrespective of time. They include class, component, deployment, object, composite structure and package diagrams.

The categorization of UML diagrams yields to a categorization of the test cases generation techniques according to the diagram(s) being used. An extra category is given to generation techniques that use other types of models rather than the UML models like the mathematical, Boolean and feature models. The categorized techniques are classified as follows:

A. Behavioral and Interactional UML Models-based Techniques

Activity diagrams can be used to derive test scenarios, a technique introduced in [31] uses a method called gray-box method [36]. The technique contains some manual steps in the algorithm of test generation. It doesn't handle fork-join efficiently and this limits the scope of the technique. It also doesn't do by all the paths; it only defines the basic paths. The fork-join structure problem was solved by the technique introduced in [37] which uses an abstraction model obtained from fully expanded activity diagrams produced by only subjecting the external inputs and outputs. The model is then converted into a flow graph that is finally used to extract test cases meeting the all-paths coverage criteria.

The use of model checking and activity diagrams is the aid of the approach proposed in [38] The UML activity diagram is translated into a formal model which is considered the NUSMV input. Next, properties in the form of CTL (Computational Tree Logic) or LTL (Linear Temporal Logic) formulas are generated using coverage criteria. Finally, the properties are applied on the NUSMV input using model checking to generate required tests. An approach initiated in [39], [40] that selects test cases from a set of randomly generated ones according to a given test criterion. A java program under testing is used to randomly generate abundant test cases. Then, by running these test cases on the program, the corresponding program execution traces are obtained. Finally, by matching these traces with the behavior of the program's activity diagram, a reduced set of test cases are selected according to the given test adequacy criterion.

Other types of diagrams have been used in many approaches to generate test cases like state chart, collaboration, and sequence diagrams. An algorithm that transforms a state chart diagram into an intermediate diagram, called the Testing Flow Graph (TFG) is shown in [22]; from the TFG it generates test cases that apply the full state and full transition coverage criteria. UML design models can be validated using test cases as well. Another approach in [41] tests the UML class, sequence, and activity diagrams used to represent a system's requirements and behavior. The expected behavior of the system under test is compared with the actual behavior that is observed during testing. The test cases generated satisfy test adequacy criteria, they are then executed on the system under test to compare its actual behavior with the expected one suggested by the UML models.

Collaboration diagrams are represented using trees in the approach presented in [13]. The approach after constructing a tree out of the system's collaboration diagram carries out a post-order traversal on it for selecting conditional predicates. Then, it applies function minimization technique to generate test data. The generated test cases

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

achieve message paths coverage as well as boundary coverage criteria. UML sequence diagrams are also used to generate test cases. Transforming them into graphs called the sequence diagram graphs (SDGs) is the first step to do so as mentioned in [42]. The presented approach then augments the SDG nodes with different information necessary to form test vectors. The test vectors are finally reformed to represent the test cases.

Altering sequence diagrams to have an initial model and making this model the starting point of the algorithm is another way of generating test cases for unit testing. It is shown in [43]. The sequence diagram is first transformed into a general unit test case model called xUnit using model-to-model transformations. Then the general xUnit model is transformed into platform specific (JUnit, SUnit etc.) test cases using model-to-text transformations. Many approaches have emerged that use more than one type of UML model to derive test cases. An approach for deriving test cases from use case and sequence diagrams is presented in [10]. It constructs a general graph called Use case Dependency Graph (UDG) from the use case diagram that shows all the use cases in the system under test. The sequence diagrams of the system are used to build flow Graphs that are used for generating test sequences. Test cases are finally generated from these test sequences using the full predicate coverage.

Another technique that uses both the use case and the sequence diagrams is shown in [44]. First, for each use case in the system under test a flow of events is specified, and then test scenarios are determined using the sequence diagrams corresponding to each use case. The flow of events and the constructed test scenarios are used together to generate the final test conditions.

Sequence diagrams can be used with activity diagrams as well to generate test cases in a strategy shown in [45] where one general sequence diagram is built for each use case. The constructed sequence diagram is then used to create several intermediate tables and flow graphs that are used in turn to create test sequences. The created test sequences are what this strategy uses to extract its final test cases. System-level test cases can be generated initially from use case models and then refined using state chart diagrams. The paper [11] introduces this methodology. An XML-based tool is used to carry out the necessary model transformations. It uses state charts and use case diagrams as well as usage graphs and usage models. It applies the minimal arc coverage technique on the usage models. The resulting test cases can run either manually or by using test tools.

Testing the interactions among model classes can be enhanced by the technique presented in [46]. The presented technique transforms the UML collaboration diagrams of the system under test and their corresponding state chart diagrams into an intermediate model called State COllaboration TEst Model (SCOTEM). It is a graph-based model used to generate test paths. But the SCOTEM has a weak point; it only deals with flat objects, the objects that change the states of others are beyond its scope. So, the model is altered in [47] so that it could work for interacting objects as well. A tool that constructs the SCOTEM is implemented and is called State COllabOration Testing EnviRonment (SCOOTER).

Scenarios and contracts have their share in generating test cases as well. An approach that uses them for generating test paths that can be used in system testing is demonstrated in [48]. The approach accepts the basic scenario and all the alternative scenarios of a use case. Then it puts the scenarios in the form of a diagram called the Interaction Overview Diagram (IOD). A transition system, called the Contracts Transition System (CTS) is then constructed by intensifying the operations in the IOD using contracts. CTS is used to generate the test paths by applying path traversals from the initial node to a final node. Also [49], [50] proposes a technique to extract test cases from scenarios that have been validated by customers using a prototype system. A different type of approaches uses mealy machines to create a formal specification of test cases that can be further used in generating test cases [51]. A mealy machine is a finite-state machine whose output values are determined both by its current state and by the values of its inputs [52], [53]. More specifically this technique analyzes and classifies the requirements and then creates a class that holds the test case specifications.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

B. *Structural UML Models-based Techniques*

Class and object diagrams are used in [54] to generate test cases. The presented methodology accepts the application code as input and runs it to create a list called the class list which contains features of classes mentioned in the application; it then uses this class list to extract the features of each class as well as the relationships between them. Finally test cases are generated based on these features and relationships. Another approach presented in [55] uses class, object, and state diagrams to define models written in a tool language called the Intermediate Format (IF). Descriptions written in IF can be animated, verified, and used to generate tests.

Class diagrams and state machines are used in [56] to generate test cases that can identify the impact of changes made in class diagrams on the corresponding state machines and in turn on the test suite. The introduced methodology assumes the presence of test suite for the program under test. It presents a UML based selective regression testing strategy to identify changes and classify them. The changes are then classified as class-driven (obtained from class diagram) and state-driven (obtained from state machine). These changes are finally used to identify fault-revealing test cases.

The paper [57] introduces the main seed of a class diagram-based methodology that generates test cases for regression testing [58]. The former paper presents a control flow analysis methodology for sequence diagrams, which is based on defining formal mapping rules between metamodels. Then Object Constraint Language (OCL)-based mapping is made between sequence diagrams and Control flow graphs called Concurrent Control Flow Graphs (CCFGs), so as to ensure the completeness of the metamodels and allow their verification. This methodology is extended to fulfill the purpose of regression testing where class diagrams are included to get more information. The current CCFG is renamed as Extended CCFG (ECCFG). The ECCFG is constructed using a sequence diagram and the corresponding class diagram. The extended methodology works by first having two versions of the same ECCFG then comparing them to identify the changed nodes and arcs which are further used as input for test case selection and generation.
Class diagrams and Object Constraint Language (OCL) are used in [59] to extract test cases from functional specifications by first mapping them to XML. Then extract specifications and use them to construct a class's hierarchy table which is used to create a classification Tree. The tree is finally pruned to extract the final test cases.

III. TEST CASES REDUCTION AND OPTIMIZATION TECHNIQUES

Reduction of the number of test cases was a major target of some approaches such as the work presented in [32], [33]. The former approach is an evolutionary-based algorithm that presents a novel model-based test suite optimization technique involving UML activity diagrams by explicating the test suite optimization problem as an Equality Knapsack Problem. The latter technique uses an algorithm depends on various testing techniques which are: Evolutionary testing, Genetic algorithms, and the Search based testing. It covers the branch coverage of functions as a unit testing model. Another technique presented in [34] proposes a requirement prioritization process during a test case generation process by introducing a method that generates multiple test suites while minimizing the number of test cases in them using UML scenarios.

A model-based regression testing approach that uses Extended Finite State Machine (EFSM) is presented in [90]. It is used to reduce the regression test suites. The modified parts of the model are tested using selective test generation techniques, but still the size of regression test suites may be very large. As a result, the approach automatically identifies the differences between the original model and the modified model as a set of elementary model modifications. For each elementary modification, regression test reduction strategies are used to reduce the regression test suite based on EFSM dependence analysis.

Dynamic symbolic execution is a structural testing technique that systematically investigates feasible paths of the program under test by running the program with different test inputs. Its main goal is to find a set of test inputs that lead to the coverage of particular test targets. Many techniques include Dynamic Symbolic Execution (DSE) technique

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

in test case generation. However, these DSE techniques, as claimed by [91], cannot generate high-covering test inputs for programs that use complex regular expressions due to the huge search space. To handle this problem, an approach is proposed in [91] named Reggae that reduces the search space of DSE in test generation, thus generating test data with higher branch coverage. However practically the number of feasible paths explored may explode, thus another search strategy called Fitnex was proposed in [92] that uses state-dependent fitness values which are computed using a fitness function to guide the path exploration.

A technique is introduced in [93] where a tool called ReAssert is built to repair test cases that have failed due to changes that have been made in the requirements which cause changes in the code. It makes changes to the test case's code to enable the passing of failed tests. It also displays the repaired and failing test code for the user to confirm the changes or make further modifications on them. However ReAssert has some limitations, like its ability to only repair about 45% of failures in open-source applications. Also ReAssert suggests a suboptimal repair, which means that a more useful repair can be possible. Moreover, if a failing test modifies expected values, creates complex expected objects, or has multiple control-flow paths, then ReAssert cannot determine what expected values need to be changed and in what way. Then comes a modification on the ReAssert in [94] to introduce a symbolic test repair which repairs more test failures and provides better repairs. It is a technique that uses the symbolic execution to change the literals in the test code. This technique can overcome some of ReAssert's limitations mentioned previously. It is also developed in java. Pex is another tool which can be used for the same aim but it is developed for .Net applications [95].

IV. CONCLUSION

A discussion on the core test processes related approaches, which emerged during the last decade, has been covered in this paper. The test cases extraction and the test data generation being the main building blocks in any testing process made them acquire recently great attention by many researchers trying to automate them in order to increase their benefits. This paper discusses many methodologies for generating test cases from UML models, whether behavioral, interactional or structural models, as well as other different types of models. It also covers many test data generation techniques based on evolutionary testing, genetic algorithms, and other more. Furthermore, several techniques for optimizing or reducing test cases are discoursed. Implementation of any of the test case generation, test data extraction or the optimization and reduction techniques can be future work. As well as applying those techniques in industry and creating tools that can automate them. Moreover, comparisons between those different techniques can be done to show the differences between them or prove the effectiveness of some techniques over others.

REFERENCES

[1] Berenbach B, Paulish D, Kazmeier J, Rudorfer A. Software and Systems Requirements Engineering in practice. The McGraw-Hill Companies Inc.: USA, 2009.
[2] Everett GD, McLeod R, Jr. Software Testing: Testing across the Entire Software Development Life Cycle. IEEE press, John Wiley & Sons, Inc., Hoboken: New Jersey; 2007.
[3] Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM, Patton GC, Horowitz BM. Model-Based Testing in Practice. Proceedings of the 21st international conference on Software engineering: New York, USA, 1999.
[4] Dias-Neto AC, Subramanyan R, Vieira M, and Travassos GH. A Survey on Model-based Testing Approaches: A Systematic Review. Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE): New York, USA, 2007.
[5] Dias-Neto AC, Travassos GH. Model-based testing approaches selection for software projects. Journal of Information and Software Technology, Butterworth-Heinemann Newton: MA, USA, 2009.
[6] Hasling B, Goetz H, Beetz K. Model Based Testing of System Requirements using UML Use Case Models. Proceedings of the International Conference on Software Testing, Verification, and Validation, IEEE Computer Society Washington: DC, USA, 2008.
[7] Object M. Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/ [2 January 2011].
[8] Swain SK, Mohapatra DP, and Mall R. Test Case Generation Based on Use case and Sequence Diagram. International Journal of Software Engineering, IJSE, 2010.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 4, April 2017

[9] Riebisch M, Philippow I, Götze M. UML-Based Statistical Test Case Generation. Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Springer-Verlag London: UK, 2003.
[10] Kansomkeat S and Rivepiboon W. Automated-Generating Test Case Using UML State chart Diagrams. Proceedings of the 2003 annual research conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT), Republic of South Africa, 2003.
[11] Samuel P, Mall R, and Kanth P. Automatic test case generation from UML communication diagrams. Information and Software Technology Journal, Butterworth-Heinemann Newton: MA, USA, 2007.
[12] Booch G, Rumbaugh J, and Jacobson I. The Unified Modeling Language User Guide. Addison-Wesley, 1999.
[13] Mitchell M. An Introduction to Genetic Algorithms. MIT Press, Cambridge, 1996.
[14] Lazić L and Medan M. Software Quality Engineering versus Softwaren Testing Process. The Telecommunication Forum (TELFOR) journal: Beograd, 2003.
[15] Smolander K. Quality Standards in Business Software Development. Master of Science Thesis, Lappeenranta University of Technology, Department of Information Technology, 2009.
[16] Graham D, Veenendaal E, Evans I, Black R. Foundations of Software Testing ISTQB Certification. International Software testing Qualifications Board, 2010.
[17] IEEE standard for software test documentation, IEEE Std 829-1998.

BIOGRAPHY

Chayashree G, Assistant professor in the Information Science and Engineering Department, GSSS Institute of Engineering and Technology for Womens, Mysuru, Karnataka, India. Completed Master of Technology (MTech) degree in 2013 from EPCET, Benagluru, Karnatak, India. .