Test Case Generation and Test Data Extraction Techniques

Pakinam N. Boghdady, Nagwa L. Badr, Mohamed Hashem and Mohamed F.Tolba

Abstract— 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 emerged during the past decade regarding the generation of test cases and test data from different models as an emerging type of model based testing. Unified Modeling Language UML models took the greatest share from among those models. Many researchers had paid a great attention to optimizing and reducing the test cases rather than generating them; light is shed on some of those approaches as well.

Index Term— Model based Testing (MBT), Evolutionary testing, Genetic algorithms, Metamorphic testing and Meta-heuristic search techniques.

I. INTRODUCTION

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.

Quality of test cases depends on how well they cover the functionalities of the system under test [16], [17] and not only on their form [18]. The test cases should be validated against known quality standards [19], [20], [21] which determine their acceptable form as well as the degree of their functional coverage which in turn specifies their level of applicability. Many metrics have emerged and are being used to measure the quality of the test cases being generated like the time, cost, effort, complexity of generation, coverage criteria and many others [22]-[26].

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.

Optimizing or even improving the quality of the test cases can be the aim of some researchers [32], [33], [34]. It can take several forms, such as decreasing the testing effort or time, decreasing the complexity or cost of the generation algorithms, increasing the functionality coverage as well as other quality and reliability issues. Also reducing the generated test cases or test data can be a form of optimization. Many tools and frameworks have been implemented; some are used for test cases generation, others used for test data extraction, whereas others are for reducing already existing sets of test cases or test data by selecting subsets from them, trying by this to optimize the use of test cases or test data.

The paper is structured as follows, a discussion of some test cases generation techniques that use behavioral, interactional and structural UML models. Next, a manifest of some test cases generation techniques that use other types of models.
Different algorithms for test data generation are demonstrated next. Then, some tests optimization and reduction techniques are discussed followed by concluding remarks.

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 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 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.

### 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.

C. Different Models-based Techniques

An automatic model-driven technique is presented in [60] to generate test cases for Graphical User Interface-based applications (GUIs). The technique uses feedback from the execution of an initial test suite, which is generated using an existing structural event-interaction graph model of the GUI. During its execution, the run-time effect of each event on all other events determines event-semantic interaction (ESI) relationships, which are used to generate new test cases.

Software Product Lines (SPLs) which are techniques for creating a collection of similar software systems from a shared set of software assets [61] requires a type of model that can define commonalities and differences between the products which have interactions among them rather than being independent from each other [62]. This type of model is called a feature model. In order to test these types of product lines, test cases that cover all possible T feature interactions (T-wise) should be generated. T-wise immensely reduces the number of test products and still ensure fair SPL coverage. Automatic generation of test cases that satisfy T-wise using SAT solvers is introduced in [63]. Another approach is presented in [64] it proposes a toolset using Alloy to generate test cases that satisfy T-wise from SPL models. (Alloy is formalism for lightweight formal analysis [65]. It provides a set of concepts that can specify elements and constraints).

Mathematical models can be used as well to generate test cases even if other models will be involved. A technique that generates test case from UML state charts using a mathematical basis is what [66] proposed. The generation algorithm is written in a language which is a mix of process algebra and a simplified version of the lambda calculus. The final test cases might be represented again as state charts or sequence diagrams or just as code in a proper programming language. No assumption about their form is determined.

A process called “2D-4A-4D” is introduced in [67]. The technique used contains two main D processes: Definition and Design. Four sub-processes, called the 4A, are also included which are: Analyze requirement specification, Analyze design diagrams, Analyze source code and Analyze type of testing. Finally four sub-processes, called 4D, are defined which are: Design test scenario, Design input data, Design test sequence and Design other elements in the set of test case. Each process and sub-process is explained in details until test cases are generated. This method was evaluated against [68], [69], [70] and as a result, it showed that it performs better at generating the smallest size of test cases.

III. TEST DATA EXTRACTION

Another technique called the meta-heuristic search has been a burgeoning interest for many researchers in recent years [71]. They are high-level frameworks, which seek solutions for combinatorial problems at a reasonable computational cost; meta-heuristic search techniques have been applied to automate test data generation in many areas. Many approaches have generated test data for structural, functional, and nonfunctional testing through the use of meta-heuristic techniques. It includes the evolutionary testing, genetic algorithms, search based testing and many others [72]-[74].

The evolutionary testing [71] is used for the automation of structural test case design which searches test data that can fulfill given structural test criteria by means of evolutionary computation. Using an evolutionary test environment is another approach to generate test data for structural test methods and it is presented in [72], [74].

Genetic Algorithms are also used to generate test data in [71] by dealing with the test-data generation problem as a function minimization problem, which allows the genetic algorithm to be applied. Also [73] introduces a tool called Genetic Algorithm Data Generation Tool (GADGET) that executes a program using a seed input; this input satisfies many of the test requirements. The initial execution of the program is used to initialize a coverage table. The coverage table is then used to select a series of test requirements. For each test requirement, the genetic algorithm is initialized and attempts to satisfy the given requirement. Whenever the genetic algorithm generates an input that satisfies a new test requirement, the new test input is recorded for future use and the coverage table is updated.

Another type of comparative approaches is introduced in [75]. The approach is called BEhavioral Regression Testing (BERT). BERT needs two versions of a program to identify the behavioral differences between them through dynamical analysis. First, it generates a large number of test inputs that focus only on the changed parts of the code. Then, it runs the generated test inputs on both the old and new versions of the code and identifies differences in the tests’ behavior. Finally, it analyzes the identified differences and presents them to the developers.

The metamorphic testing is another mechanism that can be applied on the field of automated analysis of feature models; it is also considered a way to address many testing procedures problems especially those faced by the testers in deciding whether the output of a program is correct or not. A Feature Model (FM) is “a compact representation of all the products of a software product line” [61]. The idea behind the metamorphic testing in using it to address those problems is its
ability to generate new test cases based on existing test data. The expected output of the new test cases can be checked by using what we call metamorphic relations discussed in [76], [77]. It proved to be efficient and effective in detecting most faults in a few seconds without the need of a human testing procedure. A metamorphic testing model introduced in [78] is another model for automatic generation of test data. Given a feature model and its known set of products, the algorithm generates neighboring models and their corresponding set of products. Generated products are then inspected to obtain the expected output of a number of analyses over the models.

Extended Finite State Machines (EFSM) has proved to be a powerful approach not only in modeling and deriving test sequences but in the generation of test data as well. A technique that uses these machines is presented in [79], it considers any EFSM transition a function where the function name and input parameters are derived from the corresponding transition name and input parameters. Therefore, a set of inputs to be applied to a set of functions are called sequentially is considered a test data path. A fitness function is required to guide the search for a suitable set of inputs in this technique.

State-based specifications are used to present general criteria for generating test inputs as introduced in [80]. The technique parses specifications into a general graph called specification graph, then generates test requirements for a certain criterion or a set of criteria. It then generates for each test requirement test specifications that consist of prefix values, test case values, verify conditions, exit conditions, and expected outputs to be used finally in generating actual test values that require solving algebraic equations.

An attempt to apply the concept of partition analysis for Object Constraint Language (OCL) specification of components and objects on the test data generation is shown in [81]. The OCL [82], is “an expression language that enables constraints to be described on object-oriented models”; whereas a constraint is “a restriction on one or more values within an object-oriented model”. The basic idea here is to utilize the mathematics inherent in the specification of assorted constraints to generate test data.

Search based techniques were proposed in [83] to generate test data for Simulink models. Simulink is “a software package for modeling, simulating, and analyzing system-level designs of dynamic systems”. The same authors proposed a technique that successfully generates input test data for structural and mutation testing for Simulink models [84]. Another technique proposed in [85] to address the state problem in search based testing. An extension of the above work was done in [86] to suggest investigating the application of Genetic Algorithms (GAs) on Simulink models to generate test data and then statistically compares the results to the existing work.

Flow graphs are used to generate test data in [87]. A test model called WATM is introduced. It captures data flow test artifacts of Web applications. In WATM, each component of a Web application is modeled as an object. The dataflow information of the Web application is captured using flow graphs. From the WATM, dataflow test cases for the Web application then can be derived based on the intra-object, inter-object, and inter-client perspectives.

Some Researchers have built tools for generating test data that apply some of the mentioned approaches as presented in [80], [88], whereas others made comparisons between those different tools to show the differences between them or prove the effectiveness of some techniques over others [86], [89].

IV. 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 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].

V. 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 reducing test cases are discarded. 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


