A TEST GENERATION METHOD BASED ON STATE DIAGRAM

[1] NICHA KOSINDRDECHA, [2]JIRAPUN DAENGDEJ

[1] Autonomous System Research Laboratory, Science and Technology, Assumption University, Thailand

[2] Autonomous System Research Laboratory, Science and Technology, Assumption University, Thailand

ABSTRACT

In general, the software testing phase takes around 40-70% of the time and cost during the software development life cycle. Software testing is well researched over a long period of time. Unfortunately, while many researchers have found an efficient test case generation methods to minimize time and cost, there are still a number of important research issues. The primarily issue that motivated this study is to: consume a great amount of time and cost to automatically generate tests from diagrams, with a huge size of tests and less test coverage. Therefore, this paper introduces an effective test sequence generation technique to minimize time, cost and size of tests while maximizing test coverage. The proposed technique aims to derive and generate tests from state chart diagram. The diagram is widely-used to describe a behavior of the system. In addition, this paper discusses and determines the best effective test generation methods that derive tests from diagrams.

Keywords: *test generation technique, generate test from state diagram, test case generation, test data generation and test sequence generation*

1. INTRODUCTION

Software testing is an empirical investigation activity conducted to provide all stakeholders with information about the quality of given software or applications. Software testing can be stated as a process of validating and verifying that a software or application: (a) has been implemented in line with its design specification (b) meets the business and technical requirements that guided its design and development and (c) works as expected.

John [36] argued that software testing is one of the most critical and important phases in software testing. For instance, “In June 1996 the first flight of the European Space Agency's Ariane 5 rocket failed shortly after launching, resulting in an uninsured loss of $500,000,000. The disaster was traced to the lack of exception handling for a floating-point error when a 64-bit integer was converted to a 16-bit signed integer”. This has proven that software testing is one of the most critical phases and cannot be ignored. Bertolino [7] argued that “Test case generation is a most challenging and an extensively researched activity”. Many test case generation techniques have been proposed in order to facilitate generation and preparation of test cases, such as Salas [50], Offutt [9], Heumann [34] and Turner [19]. In addition, Kaner [16] listed the purposes of test cases, for instance to find defects, maximize bug count and help managers make go / no-go decisions. These papers have shown that test cases and methods are one of the most challenging processes during software testing phase.

Beizer argued that “Software testing accounts for 50% of the total cost of software development” [14]. Many researchers from 1990 to 2006 mentioned that automated test case generation is one approach to reducing cost. Many methods have been proposed to identify a set of test cases, such as Sanjai’s work [60], Hyungchoul’s work [28] and Peter Frohlich’s work [51].

Although many test generation techniques have been proposed, there are still outstanding research issues. Specially, existing test sequence generation methods consume a huge amount of cost and time with less testing coverage. These issues may cause the following critical problems: (a) the project budget may be overrun, particularly in the large software systems (b) software testing phase may cause a delay of developing software systems and (c) some test cases may not be covered and tested properly, which causes to many known defects. As

a result, there are available rooms to improve the ability to generate tests. Therefore, this paper concentrates and introduces a new test generation method, called "TGfMMD", which aims to minimize cost and time while maximizing testing coverage.

Section 2 discusses the comprehensive set of test generation techniques. Section 3 describes outstanding issues motivated this study. Section 4 introduces a new generation process to prepare and generate tests. In addition, section 4 proposes a new effective test generation technique. The proposed method is developed to generate tests from widely used extended state diagram. Section 5 describes an experiment design and measurement metrics used in the evaluation experiment. Section 5 also describes a result and provides a short discussion of the evaluation. Section 6 concludes the contribution of this paper and provides future works for further researches. The last section represents all source references used in this paper.

2. LITERATURE REVIEW

This section surveys and describes the waterfall software development life cycle, software testing process, test case generation process and all recent research of test case generation techniques.

According to the waterfall software development life cycle (SDLC) below, basically there are five phases in the cycle, which are: (a) requirements (b) design (c) implementation (also known as development) (d) verification (also known as software testing) and (e) maintenance.

Software testing phase is the process of executing a program or system with the intent of finding errors [44]. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [26]. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Obviously, software testing is an essential activity in the SDLC. In the simplest terms, it provides quality assurance by observing the execution of a software system to validate whether it behaves as intended and to identify potential malfunctions. Testing is also widely applied by directly scrutinizing the software to provide realistic feedback of its behavior. Earlier studies estimated that testing can consume fifty percent, or even more, of the development costs [14], and a recent detailed survey in the United States [48] quantified the high economic impacts of an inadequate software testing infrastructure.

The following paragraphs describe the general process of running software testing activities. This study includes the software testing process provided by Pan [49] from Carnegie Mellon University, as follows.

1. Requirements analysis: Software testing should begin in the requirements phase of the SDLC. Software testing engineers should play a major role during the requirement phase. During the design phase, software testing engineers work with developers in determining what aspects of a design are testable and with what parameters those tests work.

2. Test planning: Test strategy, test plan, testbed creation. A testbed is a platform for experimentation for large development projects. Testbeds allow for rigorous, transparent and replicable testing of scientific theories, computational tools, and other new technologies.

3. Test development: Develop test procedures, design test scenarios, produce test cases, prepare test datasets, and build test scripts to use in testing software.

4. Test execution: Once the test plan and test cases, including test data, are generated and prepared, software testing engineers can execute the software based on the plans and tests and report any errors found to the development team.

5. Test reporting: When the test cases have been run, software testing engineers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

6. Test result analysis (also known as defect analysis): This step is done by the testing team. It is usually done along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with at a later time.

7. Retesting the resolved defects: When a defect has been resolved by the development team, the test must be run again.

8. Regression testing: In general, it is common to have a small test program built based on a subset

of tests, for each integration of new, modified or fixed software, in order to ensure that the latest delivery has not ruined anything. Additionally, this step ensures that the software product as a whole is still working correctly.

9. Test Closure: When the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

2.1 Test Case Generation Techniques

Test case generation has always been fundamental to the testing process. Bertolino [7] articulated that the test case generation step is one of the most challenging and extensively researched activities of software testing. There are many types of test case generation techniques [46] such as specification-based techniques, sketch diagram-based techniques and source code-based techniques. In addition, there are many researchers who have investigated generating a set of test cases for web-based applications [23], [75], [76]. Other techniques include goal-oriented and random approaches.

Random techniques determine a set of test cases based on assumptions concerning fault distribution. Source code-based techniques generally use a control flow graph to identify paths to be covered and generate appropriate test cases for those paths. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken.

Incorrect interpretations of complex software from non-formal specification can result in incorrect implementations leading to the necessity to test them for conformance to its specification [69].

Modeling languages may be used in the software design specification. Since UML is the most widely used language, many researchers are using UML diagrams such as state diagrams, use-case diagrams and sequence diagrams to generate test cases and this has led to sketch diagram-based test case generation techniques.

This section introduces a new “3S” classification of test case generation techniques, as follows.

1. Specification-based techniques
2. Sketch diagram-based techniques
3. Source code-based techniques

Each group can be described in details as follows.

2.1.1 Specification-Based Techniques

Specification-based techniques are methods to generate a set of test cases from specification documents such as a formal requirements specification [27], [42], [50], [57], [59], [65], [73], [74], [75]. In fact, the specification precisely describes what the system is to do without describing how to do it. Thus, the software test engineer has important information about the software’s functionality without having to extract it from unnecessary details. The advantages of this technique include that the specification document can be used to derive expected results for test data, and that tests may be developed concurrently with design and implementation. The latter is also useful for breaking “Code now test later” practices in software engineering, and for helping develop parallel testing activities for all phases [42]. The specification requirement document can be used as a basis for output checking, significantly reducing one of the major costs of testing. Specifications can also be analyzed with respect to their testability [6]. The process of generating tests from the specifications will often help the test engineer discover problems with the specifications themselves. If this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during development also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Test generation can be independent of any particular implementation of the specifications [9].

Furthermore, the specification-based technique offers a simpler, structured, and more formal approach to the development of functional tests than non-specification based testing techniques do. The strong relationship between specification and tests helps find faults and can simplify regression testing. An important application of specifications in testing is to provide test oracles. The drawbacks of the specification-based technique with formal methods are: (a) the difficulty of conducting formal analysis and the perceived or actual payoff in project budget. Testing is a substantial part of the software budget, and formal methods offer an opportunity to significantly reduce testing costs, thereby making formal methods more attractive from the budget perspective [27] and (b) there is greater manual effort or processes in generating test cases, compared with techniques involving

automatic generation processes. This research reveals that many techniques have been proposed such as heuristics algorithms [38], [65], model checkers [27], [57], [60] and hierarchy approaches [42], [74], [75]. The following paragraphs describe examples of existing specification-based techniques that have been proposed since 1997.

Percy [50] presented the underlying theory by providing a set of test cases with formal semantics and translated this general testing theory to a constraint satisfaction problem. A prototype test case generator serves to demonstrate the automation of the method. It works on Object Constraint Language (OCL) specifications. The OCL is part of the UML 2.0 standard. It is a language allowing the specification of formal constraints in context of a UML model. Constraints are primarily used to express invariants of classes, pre-conditions and post-conditions of operations. These invariants become elements of test cases. In their work, they aimed to generate test-cases focusing on possible errors during the design phase of software development. Examples of such errors might be a missing or misunderstood requirement, a wrongly implemented requirement, or a simple coding error. In order to represent these errors, they introduced faults into formal specifications. The faults are introduced by deliberately changing a design, resulting in wrong behavior possibly causing a failure. They focused dedicatedly on the problem of generating test cases from a formal specification. The problem can be represented as a Constraint Satisfaction Problem (CSP). A CSP consists of a finite set of variables and a set of constraints. Each variable is associated with a set of possible values, known as its domain. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. Formally, the conjunction of these constraints forms a predicate for which a solution should be found. To resolve the above problem, they proposed to embed the test generation problem modeled as a CSP into a specially designed and implemented Constraint System. But this is not a novelty because this approach has been widely explored and implemented. The novelty in their approach is the relation that they formalized between fault-based testing and constraint solving.

Miao [42] presented a framework based on Phil Stocks and David Carrington's work [85], [86] He defined a test class using an object-oriented concept instead of Phil Stock's test template in the framework. Phil Stock's test template defines test data only. The benefit of their test framework for Z specifications is that the test data and oracles are defined in a test class which also contains the information of before states and after states for an operation. The test framework is therefore a dynamic system involving state change, containing three components: (a) valid input space & output space (b) test class & test state space and (c) test class hierarchy & instantiation.

Jefferson [10] presented a model for developing test inputs from state-based specifications, and formal criteria for test case selection. For state-based specification technique, their paper used the term specification-based testing in the narrow sense of using specifications as a basis for deciding what tests to run on software. Their proposed approach is related to Blackburn's state-based functional specifications of the software, expressed in the language, T-Vec [45]. It is used to derive disjunctive normal form constraints, which are solved to generate tests. Also, their approach is related to Weyuker, Goradia [22] who presented a test case generation method from Boolean logic specifications. Moreover, they introduced several criteria for system level testing. These criteria are expected to be used both to guide the testers during system testing and to help the testers find rational, mathematical-based points at which to stop testing. In those criteria, tests are generated as multi-part, multi-step and multi-level artifacts. The multi-part aspect means that a test case is composed of several components: test case values, prefix values, verify values, exit commands, and expected outputs. The multi-step aspect means that tests are generated in several steps from the functional specifications by a refinement process. The functional specifications are first refined into test specifications, which are then refined into test scripts. The multi-level aspect means that tests are generated to test the software at several levels of abstraction.

Cunning [66] was interested in the model-based codesign of real-time embedded systems. It relies on system models at increasing levels of fidelity in order to explore design alternatives and to evaluate the correctness of these designs. As a result, the tests that they desire should cover all system requirements in order to determine if all requirements have been implemented in the design. The set of generated tests is maintained and applied to system models of increasing fidelity and to the system prototype in order to verify the consistency between models and physical realizations. In the

codesign method, test cases are used to validate system models and prototypes against the requirements specification. In the paper, they presented continuing research toward automatic generation of test cases from requirements specifications for event-oriented, real-time embedded systems. They used a heuristic algorithm to automatically generate test cases in their works. The heuristic algorithm uses the greedy search method followed by a distance based search if needed. The algorithm with pseudo code is addressed in their paper [65].

Hung [27] focused on existing research in using model checking to generation test cases. He touched on several areas, like the methodology of properly testing software, the use of model checking to generate tests suits and specialization of specification to suit the needs of test generation. A model checker is used to analyze a finite-state representation of a system for property violations. If the model checker analyzes all reachable states and detects no violations, then the property holds. However, if the model checker finds a reachable state that violates the property, it returns a counterexample – a sequence of reachable states beginning in a valid initial state and ending with the property violation. In his technique, the model checker is used as a test oracle to compute the expected outputs and the counterexamples it generates are used as test sequences. In summary, his approach is used to generate test cases by applying mutation analysis. Mutation analysis is a white-box method for developing a set of test cases which is sensitive to any small syntactic change to the structure of a program.

Sanjai [61] presented a method for automatically generating test cases to structural coverage criteria. He showed how, given any software development artifact that can be represented as a finite state model, a model checker can be used to generate complete test cases that provide a predefined coverage of that artifact. He provided a formal framework that is: (a) suitable for defining their test-case generation approach and (b) easily used to capture finite state representations of software artifacts such as program code, software specifications, and requirements models. He showed how common structural coverage criteria can be formalized in their framework and expressed as temporal logic formulae used to challenge a model checker to find test cases. Finally, he demonstrated how a model checker can be used to generate test sequences for modified condition and decision (MC/DC) coverage. Their approach to generating test cases involves using the model-checker as the core engine. A set of properties called trap properties [8], is generated and the model-checker is asked to verify the properties one by one. These properties are constructed in such a way that they fail for the given system specification.

2.1.2 Sketch diagram-Based Techniques

Sketch diagram-based techniques are methods to generate test cases from model diagrams like UML Use Case diagram [11], [34], [35], [39] and UML State diagrams [2], [4], [5], [20], [25], [30], [37], [68]. The following paragraphs survey current sketch diagram-based test case generation techniques that have been proposed for traditional and web-based application for a long time. A major advantage of model-based V&V is that it can be easily automated, saving time and resources. Other advantages are shifting the testing activities to an earlier part of the software development process and generating test cases that are independent of any particular implementation of the design [11]. The following paragraphs describe examples of existing specification-based techniques that have been proposed since 2000.

Jim [34] presented how using use cases to generate test cases can help launch the testing process early in the development lifecycle and also help with testing methodology. In a software development project, use cases define system software requirements. Use case development begins early on, so real use cases for key product functionality are available in early iterations. According to the Rational Unified Process (RUP), a use case is used to fully describe a sequence of actions performed by a system to provide an observable result of value to a person or another system using the product under development. Use cases tell the customer what to expect, the developer what to code, the technical writer what to document, and the tester what to test. He proposed three-step process to generate test cases from a fully detailed use case: (a) for each use case, generate a full set of use-case scenarios (b) for each scenario, identify at least one test case and the conditions that will make it execute and (c) for each test case, identify the data values with which to test.

Johannes [35] raised the practical problems in software testing as follows: (a) lack of planning/time and cost pressure, (b) lack of test documentation, (c) lack of tool support, (d) formal language/specific testing languages required, (e) lack of measures, measurements and data to

quantify testing and evaluate test quality and (f) insufficient test quality. Their proposed approach to resolve the above problems is to derive test cases from scenarios / UML use cases and state diagrams. In their work, the generation of test cases is done in three stages: (a) preliminary test case and test preparation during scenario creation (b) test case generation from Statechart and dependency charts and (c) test set refinement by application dependent strategies (intuitive, experience-based testing).

Manish [39] were interested in testing web based applications. Web based applications are of growing complexity and it is a serious business to test them correctly. They focused on black box testing which enables the software testing engineers to derive sets of input conditions that will fully exercise all functional requirements. They believed that black box testing is more generally suitable and more necessary for web applications than other types of application. Furthermore, they proposed four steps to generate test cases, based on Heumann's four-steps [34], as follows: (a) prioritize use cases based on the requirement traceability matrix (b) generate tentatively sufficient use cases and test scenarios (c) for each scenario, identify at least one test case and the conditions and (d) for each test case, identify test data values. They also presented that the test cases contains: a set of test inputs, execution conditions and expected results developed for a particular objective.

Avik [5] described a new model based testing technique developed to identify critical domain requirements. The new technique is based on modeling the system under test using a strongly typed domain specific language (DSL). In the new technique, information about domain specific requirements of an application are captured automatically by exploiting properties of the DSL and are subsequently introduced in the test model. The new technique is applied to generate test cases for the applications interfacing with relational databases and the example DSL. Test suites generated using the new techniques are enriched with tests addressing domain specific implicit requirements.

Valdivino [69] focused on test sequence generation from a specification of a reactive system, space application software, in Statecharts [24] and the use of PerformCharts [70]. In order to adapt PerformCharts to generate test sequences, it has been associated to a test case generation method, switch cover, implemented within the Condado tool [3]. Condado is a test case generation tool for FSM. The algorithm implemented in Condado is known as sequence of "de Bruijn". The steps in the algorithm are: (a) a dual graph is created from the original one, by converting arcs into nodes (b) by considering all nodes in the original graph, where there is an arc arriving and another arc leaving, an arc is created in the dual graph (c) the dual graph is transformed into a "Eulerized" graph by balancing the polarity of the nodes and (d) finally, the nodes are traversed registering those that are visited.

2.1.3 Source Code-Based Techniques

Source code-based techniques generally use control flow information to identify a set of paths to be covered and generate appropriate test cases for these paths. The control flow graph can be derived from source code. The result is a set of test cases with the following format: a) test case ID b) test data c) test sequence (also known as test steps) d) expected result e) actual result and f) pass / fail status. The following paragraphs describe the source code-based techniques that have been proposed since 1999.

Sami [59] presented a novel approach to automated test case generation. Several approaches have been proposed for test case generation, mainly random, source code-based, goal-oriented and intelligent approaches [58]. Random techniques determine test cases based on assumptions concerning fault distribution (e.g. [1]). Source code-based techniques generally use control flow information to identify a set of paths to be covered and generate appropriate test cases for these paths. These techniques can further be classified as static or dynamic. Static techniques are often based on symbolic execution e.g. [18], whereas dynamic techniques obtain the necessary data by executing the program under test e.g. [12]. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken e.g. [58]. Intelligent techniques of automated test case generation rely on complex computations to identify test cases e.g. [47]. Another classification of automated test case generation techniques can be found in [47]. Their algorithm proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that in [12]. The path to be covered is considered step-by-step, i.e. the goal of covering a path is divided into sub-goals and test cases are then searched to fulfill them. The search process, however, differs substantially. In Bogdan's work [12], the search process is conducted according to a specific error function. In their approach, test cases are determined using binary search, which requires

certain assumptions but allows efficient test case generation.

David [19] proposed an activity oriented approach. Their approach is one possible approach to test web applications; it is a black-box test based on user interactions with the web application. As web applications become more sophisticated, the functionalities of web pages have become more intricate, convoluted and loaded with links, buttons, and multiple forms. Manual testing of such web applications, though unavoidable, is grueling and often not reliable. Hence it is preferable to develop automated tests that can expose failures and deviations from intended behavior. The user interactions may be as simple as clicking a button or as complicated as filling several forms to accomplish a task. Such likely user interactions are identified, analyzed, and defined to build an activity oriented testing model. This test model can be applied to functional testing and load testing. It can also be used for data building (populating the application with data) for the purpose of manual testing and intermediate client evaluations. An activity test program utilizes the test model suitably for the above mentioned concerns and generates a test report. A test report comprises a list of tests and statuses, which is one of passed, failed or unreachable.

2.2 Test Data Generation Techniques

Ian [29] included a test data generation process, known as "preparing test data", is one of the important activities in the software testing process. Phil [52] stated that a test data generation technique is one of the interesting research topics with many available research issues. This section discusses existing techniques to prepare and generate a set of input and output data, along with limitations. There are many researchers who studied and proposed effective test data methods, such Hayes's works [32], Grindal's work [40], Richard's works [57], Sasa's techniques [62] and Sara's case studies [63].

This section introduces a new "2S" classification of test data generation techniques, as follows: (a) specification-based techniques and (b) source code-based techniques (also known as path-oriented test data techniques). Each group can be described in details as follows.

2.2.1 Specification-Based Techniques

Specification-based techniques are methods to generate test data from specification documents such as state-based specification [6], [33], [55], [56], object constraint language (OCL) and test specification language (TSL) [43]. Eventually, those techniques generate a set of test data with the following format: (a) test case ID (b) input data and (c) output data. The following paragraphs survey existing specification-based test data generation techniques studied since 1999.

Aynur [6], [33] defined the following definition in their work: (a) test requirements are specific things that must be satisfied or covered during testing and (b) test specifications are specific descriptions of test cases including test data, often associated with test requirements or criteria. They presented a test data generation method, based on Offut's state-based technique, to prepare and generate a set of data from UML state charts diagram. They proposed to use the TSL language to describe all elements of a test case, like input, output and pre-condition. However, they concentrate on the following elements: (a) pre-condition values (b) verify values (c) exit command and (d) expected output data. Generally, those elements are directly derived from triggering events and pre-conditions in the state chart diagram. The pre-condition values include all required input data. Any input data, which are required to show the results, are the verify values. The exit commands are depended on the system or program being tested. The expected output data are created from the after-values of the triggering events and post-conditions.

Jeff [33] presented a method to generate a set of input data from state-based specifications. They proposed general criteria for preparing data. The criteria include the following techniques: (a) transition predicates (b) transitions (c) pairs of transitions and (d) sequences of transitions. These techniques provide coverage criteria that are based on the specifications, and are made up of several parts, including test prefixes that contain inputs necessary to put the software into the appropriate state for the test values.

However, there are a few researchers who investigated in generating a set of data for the object-oriented modeling. For example, Mohammed Benattou, Jean-Michel Bruel [43] presented partition analysis concept, on which the approach for generating test data is based. Also, they used the OCL language to describe and generate test data. In fact, the OCL language is used in the UML semantics document to specify the well-formed rules of the UML meta-model. Also, the OCL language is a pure expression language and can be

used to specify invariants, pre-condition, post-condition, and other kind of constraint in the specification.

In addition, the literature review shows that there are no existing test data techniques used for a web-based application or complex systems, like real-time system or embedded system.

2.2.2 Source Code-Based Techniques

Source code-based techniques, known as path-oriented based techniques, are techniques to generate and prepare test data from control flow graph. The control flow graph can be directly derived from source or binary code. The literature review shows that there are a few researchers who concentrate on preparing and generating both of input and output data by using source or binary code. The following paragraphs present shortly a description of existing test data generation techniques.

Bogdan [12] presented an alternative approach of test data generation, referred to as a dynamic approach of test data generation, which is based on actual execution of a program under test, dynamic data flow analysis, and function minimization methods. Test data are developed using actual values of input variables. When the program is executed on some input data, the program execution flow is monitored. If, during program execution, an undesirable execution flow at some branch is observed then a real-valued function is associated with this branch. This function is positive when a branch predicate is false and negative when the branch predicate is true. Function minimization search algorithms are used to automatically locate values of input variables for which the function becomes negative. In addition, dynamic data flow analysis is used to determine input variables which are responsible for the undesirable program behavior, leading to significant speed-up of the search process. Bogdan mentioned that arrays and dynamic data structures can be handled precisely because during program execution all variables values, including array indexes and pointers, are known; as a result, the effectiveness of the process of test data generation can be significantly improved.

Additionally, Bogdan [13] presented an effective test data generation technique by using the control flow graph, particularly used in the regression testing phase. The technique focuses on automatically generate test data for a modified program or source code. It utilizes the original version of the program in the test data generation process. Specifically, it attempts to automatically generate an input data on which the original program and its modified version yield a different result (or also known as output).

Jane [31] presented the input validation testing (IVT) technique to prepare test data. The IVT technique has been developed to address the problem of statically analyzing input command syntax as defined in the English textual interface and requirements specifications. The technique does not require design or code. Thus, it can be applied early in the lifecycle. It focuses on the specified behavior of the system and uses a control flow graph. It contains four major aspects: (a) a way to specify the format of requirement specifications, (b) an approach to analyze an input command (c) a method to generate valid test data and (d) a technique to prepare an error test data

In addition, the literature review shows that there are a few researchers who studied in the test data generation techniques for web-based applications. For example, Chien-Hung Liu [17] extended traditional data flow testing techniques to generate a set of data for web applications.

2.3 Test Sequence Generation Techniques

This section discusses test sequence generation techniques that are used to generate a set of test steps or procedures in the test cases. The literature review reveals that there are several researchers who proposed effective methods to prepare and identify test sequence in the test case, for example, Stefania's work [64], Dalal's study [67] and Eric [73]. Also, this section introduces a new "2S" classification of existing test sequence generation techniques, as follows: (a) specification-based techniques and (b) sketch diagram-based techniques (also known as model-based test sequence techniques). Each group can be described shortly as follows.

2.3.1 Specification-Based Techniques

Specification-based test sequence generation techniques are methods to generate test sequence or test steps from requirement specifications. The literature review shows that the test sequence or test steps are one of the elements in the test cases. It also shows that there is only one researcher who studied these techniques.

Sanjai [60] proposed an effective test sequence generation technique to prepare and generate a set

of sequences used in the test case. They developed the generation method by using the hypothesis, which model checkers can be effectively used to automatically generate test steps or sequence.

2.3.2 Sketch Diagram-Based Techniques

Sketch diagram-based test sequence generation techniques are methods to derive and generate test sequences from diagrams. The literature review shows that UML diagrams are typically used to prepare a set of sequence or test steps. The basic UML diagrams that researchers have studied and used are: (a) UML activity diagram (b) UML state chart diagram and (c) UML sequence diagram.

The following shortly describes examples of existing sketch diagram-based test sequence generation techniques.

Hyungchoul [28] proposed a method to generate a test sequence, by using the UML activity diagram. The method aims to minimize the number of test steps generated while deriving all practically useful tests. It consists of three main processes: (a) build an input / output activity diagram (also known as IOAD) (b) transforms to a directed graph, from which test steps for the initial activity diagram are derived and (c) generate a set of test sequence.

Wang [72] proposed an approach to generate test sequences directly from the UML activity diagram using a gray-box method, where the design is reused to avoid the cost of test model creation. The paper shows that test scenarios are directly derived from the activity diagram modeling an operation. Therefore, all the information, such as test sequences or test data, is extracted from each test scenario. At last, the possible values of all the input/output parameters could be generated by applying a category-partition method, and test case could be systematically generated to find the inconsistency between the implementation and the design. Gray-box testing method, in the designer's viewpoint, generates test sequences based on high level design models which represent the expected structure and behavior of the software under test (SUT). The design specifications are the intermediate artifact between requirement specification and final code. Those specifications preserved the essential information from the requirement, and are the basis of the code implementation. Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test sequences which can satisfy the path conditions by black box method. It can find problems which used to be ignored by both black and white method. Gray-box method could systematically generate test sequences directly from the activity diagrams which can be used to test the system at code level.

Farooq [68] presented an effective control-flow based test sequence generation technique using the UML activity diagram, version 2.0, which is a behavioral type of UML diagram. They proposed a technique that enables the automatic generation of test sequences according to a given coverage criteria from the execution of the Colored Petri Nets model. There are three steps, which are: (a) convert the information from UML activity diagram into control-flow graph, in the format of Colored Petri Net (b) define coverage criteria from the execution of the Petri Net model and (c) generate a test sequence by using a random walk algorithm, based on the probability.

Samuel [54] proposed an automatic test sequence generation method that derived test sequence from state machine diagrams. The state machine diagrams are one of the extended UML state chart diagram. They proposed to have three main steps in the generation algorithm, which are: (a) to select a predicate on a transition from the state diagram (b) to transform to the function and (c) generate a test sequence based on the transformed function.

Samuel [52] presented an approach to generate test sequences from the UML sequence diagrams, version 2.0. UML Sequence diagrams are one of the most widely used UML models in the software industry. They found that existing test sequence generation techniques do not encompass certain important features of the UML sequence diagrams, version 2.0. Thus, they proposed an effective method to generate a set of test steps by considering many key features of UML sequence diagram, version 2.0, like loop, alt and break feature.

3. RESEARCH CHALLENGES

This section discusses the details of research issues motivated this study. The literature reviews show that there are available rooms for researchers to develop and enhance the ability to generate a set of test case, test sequences and test data from sketch diagrams, such as existing techniques ignores some important information derived from those diagrams, some techniques can't generate both of test data and test sequence from the diagrams and there are limitations for some techniques for an commercial

systems, like real-time system, concurrent system and financial banking system.

The research issues that motivated this paper are: (a) existing techniques consumes a great deal of effort, time and cost to automatically generate test cases from extended state chart diagram (b) existing techniques generate a significant number of test cases with less coverage and (c) non-effective test case generation methods for state or node coverage.

The literature review reveals that existing sketch diagram test sequence generation techniques consumes a great amount of time and cost. Some techniques indirectly derive and generate test sequences from diagrams, which it takes longer time to transform those diagrams and design tests respectively. This is one of the interesting outstanding research issues for researchers who are interested in test sequence generation methods.

Also, the study shows that existing methods typically design and generate a large set of test cases. However, even if those methods generate a greater size of tests, but those tests do not maximize test coverage. Some methods prepare and generate a significant number of tests with less test coverage. Consequently, it may cause a lot of known defects.

Finally, there are available rooms to improve the ability to generate test sequence and maximize state or node coverage. Researchers should develop a test sequence methods that minimize a size of tests, time and cost, while preserving test coverage.

4. PROPOSED METHODS

This section discussed a proposed technique that prepare and generate both of test data and test sequence from a state diagram. The state diagram is a type of diagrams used in computer science and related fields to describe the behavior of systems. State diagram requires that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction. There are many forms of state diagrams, which differ slightly and have different semantics. Many practitioners have proposed several types of those diagrams, such as Adam Petri [15], Wagner [71] and Mealy [41]. Also, they have applied these diagrams into the commercial systems. The most widely used diagrams in those systems are extended state diagrams. Thus, this paper proposes to automatically prepare and generate tests from those extended state diagrams, called "TGfMMD" method. Also, the literature reviews reveal that the most famous and widely used extended state diagrams is a "Mealy Machine" diagram. The Mealy Machine diagram is extended from the UML state diagram. Both of these diagrams are used to describe the behavior of systems but differ in the sense of Merly Machine diagram has input and output while normal state diagram doesn't have.

The following shows the flow-chart diagram of the proposed method. The method is developed for directly generating tests from Merly Machine diagram.

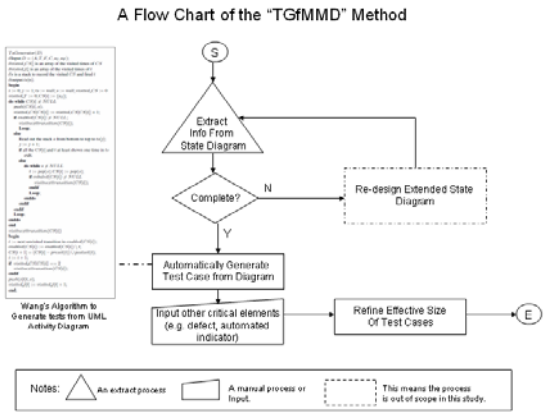

Figure 1. A Flow Chart of Test Case Generation Method

From the above figure, the exact procedures can be described shortly as follows:

The procedure begins with extracting all required information entered in the extended state diagram.

Let S = {s1, s2, …, sn} for S to be a set of states, I = {i1, i2, …, in} for I to be a set of input data, O = {o1, o2, …, on} for O to be a set of output data, T = {tr1, tr2, …, trn} for T to be a set of transitions or edges.

In this step, there is a verification process to ensure that all required information in the diagram is completed, like state id, state information, input, output and conditions. If the information is not available and completed, then the process will return false to allow re-designing the diagram and filling more information. The re-design work is out of this paper's scope. Otherwise, the process will go to next step.

In this step, the process generates a set of test case, test data and test sequence and Wang's algorithm [72]. Wang's algorithm is well-known and widely used in the industry. His algorithm is used to derive test cases from state diagram. The TGfMMD method is built based on his algorithm. It derives and generates test cases from the following sets: (S, I, O and T), as mentioned in the first step.

Let $TS = \{tc_1, tc_2, \ldots, tc_n\}$ for TS to be a collection of test cases. Thus, test case can be defined as follows: $TC = \{S, I, O, TR\}$ where S is a set of stages, I is a set of input, O is a set of output and TR is a set of transitions.

Although Wang's algorithm is widely used, but it doest not cover other critical attributes, like defect id, dependency and automated test case indicator. Thus, the TGfMMD method proposes to manually input those values.

The last step proposes to minimize a size of generated test cases while maximizing test coverage in the set of test cases. In order to generate an effective size of generated test case, this step contains two sub-tasks, which are: (a) calculate node coverage for each test case and (b) select effective test cases.

Let NodeCov (tc) = $\{t_1, t_2, \ldots, t_n\}$ for NodeCov(tc) to be a set of test cases that tc is covered by $t_1, t_2, \ldots, t_n$. Therefore, if a number of set tc is zero, then tc is included in the effective set of test cases.

The following presents an example of TGfMMD method that generates and derives a set of test cases from a mealy machine diagram.

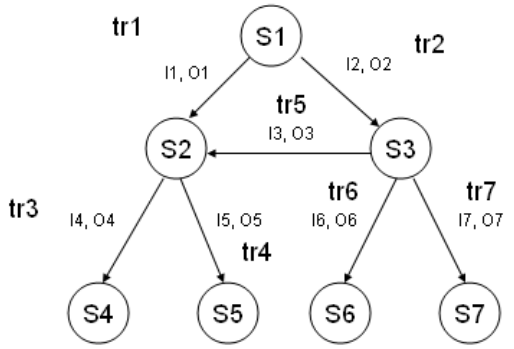

Figure 2. An Example of Mealy Machine Diagram

First, the TGfMMD method aims to extract all required information from the above diagram. Thus, the result can be:

$S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ where S is a set of stage and sn is a stage or node in the diagram.

$I = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$ where I is a set of input data and in is an input value.

$O = \{o_1, o_2, o_3, o_4, o_5, o_6, o7\}$ where O is a set of output data and on is an output value.

$T = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7\}$ where T is a set of transitions or edges and tr_n is a transition between source and destination stage.

Each transition contains both of source and destination stage or node, as follows: $tr_n = \{s_1, s_2\}$ where s_1 is a source of stage and s_2 is a destination of stage. Thus, each transition can be extracted as follows:

$tr_1 = \{s_1, s_2\}$

$tr_2 = \{s_1, s_3\}$

$tr_3 = \{s_2, s_4\}$

$tr_4 = \{s_2, s_5\}$

$tr_5 = \{s_3, s_2\}$

$tr_6 = \{s_3, s_6\}$

$tr_7 = \{s_3, s_7\}$

Second, this step is to verify the completion of extracted information, derived from the diagram. This step assumes that the diagram and information are complete in this example.

Third, the TGfMMD method is applying Wang's algorithm, in [72], to derive and generate test cases. Therefore, all tests can be generated as follows:

$TC_1 = \{s_1, s_2, i_1, o_1, tr_1\}$

$TC_2 = \{s_1, s_3, i_2, o_2, tr_2\}$

$TC_3 = \{s_1, s_2, s_4, i_1, i_4, o_1, o_4, tr_1, tr_3\}$

$TC_4 = \{s_1, s_2, s_5, i_1, i_5, o_1, o_5, tr_1, tr_4\}$

$TC_5 = \{s_1, s_3, s_2, s_4, i_2, i_3, i_4, o_2, o_3, o_4, tr_2, tr_5, tr_3\}$

$TC_6 = \{s_1, s_3, s_2, s_5, i_2, i_3, i_5, o_2, o_3, o_5, tr_2, tr_5, tr_4\}$

$TC_7 = \{s_1, s_3, s_6, i_2, i_6, o_2, o_6, tr_2, tr_6\}$

$TC_8 = \{s_1, s_3, s_7, i_2, i_7, o_2, o_7, tr_2, tr_7\}$

$TC_9 = \{s_2, s_4, i_4, o_4, tr_3\}$

$TC_{10} = \{s_2, s_5, i_5, o_5, tr_4\}$

$TC_{11} = \{s_3, s_2, i_3, o_3, tr_5\}$

$TC_{12} = \{s_3, s_2, s_4, i_3, i_4, o_3, o_4, tr_5, tr_3\}$

$TC_{13} = \{s_3, s_2, s_5, i_3, i_5, o_3, o_5, tr_5, tr_4\}$

$TC_{14} = \{s_3, s_6, i_6, o_6, tr_6\}$

$TC_{15} = \{s_3, s_7, i_7, o_7, tr_7\}$

The last step is to minimize a set of test cases by calculating node coverage for each test case and determine which test cases are covered by other test cases.

NodeCov (TC_1) = $\{TC_3, TC_4, TC_5, TC_6\}$

NodeCov (TC_2) = $\{TC_5, TC_6, TC_7, TC_8\}$

NodeCov (TC_3) = $\{TC_5\}$

NodeCov (TC_4) = {TC_6}

NodeCov (TC_5) = {}

NodeCov (TC_6) = {}

NodeCov (TC_7) = {}

NodeCov (TC_8) = {}

NodeCov (TC_9) = {TC_3, TC_5, TC_{12}}

NodeCov (TC_{10}) = {TC_4, TC_6}

NodeCov (TC_{11}) = {TC_5, TC_6, TC_{12}, TC_13}

NodeCov (TC_{12}) = {TC_5}

NodeCov (TC_{13}) = {TC_6}

NodeCov (TC_{14}) = {TC_7}

NodeCov (TC_{15}) = {TC_8}

Therefore, the following test cases should be ignored during the execution time: TC_1, TC_2, TC_3, TC_4, TC_9, TC_{10}, TC_{11}, TC_{12}, TC_{13}, TC_{14} and TC_{15}. The remaining effective set of test cases is {TC_5, TC_6, TC_7, TC_8}.

5. EVALUATION

This section describes an experiment design, measurement metrics and results in order to determine the most recommended test case generation derived from the extended state chart diagram.

5.1 Experiment Design

A comparative evaluation method has proposed in this experiment design. The high-level overview of this experiment design can be found as follows:

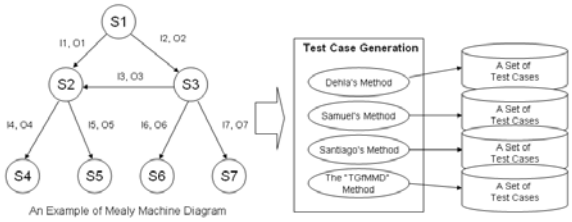

Figure 3. Experiment for Test Case Generation.

From the above figure, the following lists procedures of this experiment:

1. Prepare Experiment Data. This step is designs to generate 7 states along with 7 input data from Mealy Machine diagram. The literature review [33], [53], [54], [73] shows that other researchers use a simple diagram to evaluate their generation methods. They do not use large complex diagram used in the commercial industry, as their case study or evaluation method.

2. Generate Test Case. A comparative evaluation method has been made among the proposed test case algorithm, which are: Dehla's algorithm [21], Samuel's technique [54] and Santiago's method [69].

3. Evaluate Results. In this step, graph and discussion have been proposed to evaluate results for the previous techniques.

5.2 Measurement Metrics

The section lists the measurement metrics used in the experiment.

1. Size of Test Case: This is a total number of generated test cases by each test case generation methods described in the previous section. This experiment proposes to use the following formula to compute the percentage of size:

*% Size = (# Size / # of Total Size)*100*

Where:

- *% Size* is a percentage of a number of test cases generated by each method.
- *# of Size* is a number of test cases that each method generates.
- *# of Total Size* is a maximum number of test cases in the experiment, which is assigned to 100.

2. Percentage of Node Coverage: This is an indicator to identify a number of nodes or states that a set of test cases cover in the state chart diagram [54]. Every node in the state diagram must be tested at least one time [33], [53], [54], [73]. Thus, each method is expected to generate test cases that cover all nodes in the diagram. This experiment proposes to use the following formula to compute the percentage of node coverage in the diagram.

*% NC = (# of Node / # of Total)*100*

Where:

- *% NC* is a percentage of node coverage.
- *# of Node* is a number of nodes or states covered in the state chart diagram.
- *# of Total* is a total number of nodes in the diagram.

3. Total Time: This is a total number of times running the generation methods in the experiment. This metric is related to time used during testing development phase (e.g. design test scenario and produce test case). Therefore, less time is desirable. It can be calculated as the following formula:

Total = Preparation Time + Compile Time + Running Time

Where:

- *Total* is a total number of times consuming in running generation methods.
- *Preparation time* is a total number of times consuming in preparing before generating test cases.
- *Compile time* is a time to compile source code / binary code in order to execute program.
- *Running time* is a total time to run the program under this experiment.

Also, the following represents a formula that calculates the total time in the format of percentage.

*% Time = (# Total / # of Maximum Total Time)*100*

Where:

- *% Time* is a percentage of total time.
- *# of Total* is a total time consumed during the generation process.
- *# of Maximum Total Time* is a maximum time in the experiment, which is assigned to 100 seconds.

5.3 Results and Discussion

This section discusses an evaluation result of the above experiment. This section presents a graph that compares the TGfMMD method to other four existing test generation techniques, based on the following measurements: (a) size of test cases (b) percentage of node coverage and (c) total time. Those four techniques are: (a) Sinha's technique (b) Santiago's method (c) Reza's algorithm and (d) Shams's method. There are two dimensions in the following graph: (a) horizontal and (b) vertical axis. The horizontal represents three measurements whereas the vertical axis represents the percentage value.

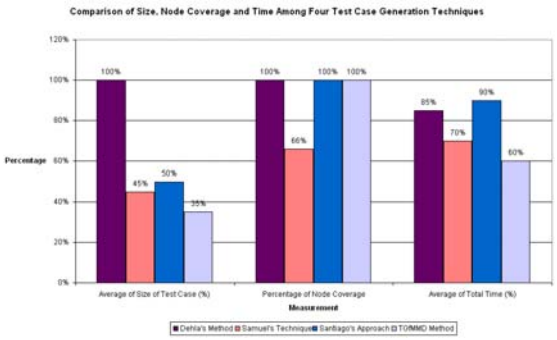

Figure 4. An Evaluation Result of Generation Methods.

The above graph shows that TGfMMD method generates the smallest size of test cases whereas Dehla's method generates the biggest size of test cases. Samuel's approach has the least percentage of node coverage comparing other techniques. Other three techniques cover 100% all nodes or state in the state chart diagram. TGfMMD method consumes a minimum of total time by 60%. Santiago's approach consumes total time greater than TGfMMD by 30%.

The following table ranks test case generation techniques used in the experiments, based on the above measurements, by 1 is the first, 2 is the second, 3 is the third and 4 is the last.

Table 1. Test Case Generation Techniques Ranking Table

Methods	Size of Test Cases	Percentage of Node Coverage	Total Time
Dehla's Method	4	1	3
Samuel's Technique	3	4	2
Santiago's Approach	2	1	4
TGfMMD Method	1	1	1

In the conclusion, the TGfMMD method is the most recommended method to generate the smallest size of test cases with the minimum total time and cover 100% all nodes in the state diagram.

6. CONCLUSION

With the existing techniques since 1990, this paper introduces a new "3S" classification of test case generation techniques, which are: specification-based technique, sketch-diagram-based technique and source code-based technique. First, the specification-based technique is a method

to generate a set of test cases from specification documents such as formal requirement specification. Second, the sketch diagram-based technique, also known as model-based technique in other papers, is method to generate test cases from model diagrams like UML Use Case diagram [34], [35], [39], UML Sequence diagram [11] and UML State diagram [2], [4], [5], [20], [25], [30], [37], [68]. Last, the source code-based technique, also known as path-oriented in other papers, generally uses control flow information to identify a set of paths to be covered and generated the appropriate tests for the paths. Also, this paper introduces a new "2S" classification of existing test data generation techniques, researched since 1990, as follows: specification-based technique and source code-based technique. First, the specification-based technique is an approach to generate a set of input and output data, along with pre-condition, derived from the requirement specifications. Second, the source code-based technique aims to design test data by using control flow graph and source code.

Moreover, this paper proposes a new "2S" classification of existing test sequence generation techniques, which are: specification-based technique and sketch diagram-based technique. The specification-based technique is a method to prepare and design a set of test steps in the test case, derived from the requirement or design specifications. The sketch diagram-based technique is an approach to generate a test sequence from UML diagrams, such as UML activity diagram, UML state chart diagram and UML sequence diagram. According to the above comprehensive literature review, this paper proposes a new test case generation process, called "2D-4A-4D". The new procedure contains two main processes: (a) define and (b) design. The first process is composed of four sub-processes, called "4A", which are: (a) analyze requirement specification (b) analyze designed diagrams (c) analyze source code and (d) analyze type of testing. The second process is also composed of four sub-processes, called "4D", which are: (a) design test scenario (b) design input data (c) design test sequence and (d) design other elements in the set of test case.

There are many research challenges and gaps in the test case generation area. Those challenges and gaps can give the research direction in this field. For example, the existing test case generation techniques generally ignore the size of test cases. As a result, it will take a longer time and effort to execute the set of test cases. Another example is that most test case generation are inefficient test case generation techniques. Those techniques do not concern the limitation, such as time, cost and effort. However, the research issues that motivated this paper are: (a) existing techniques consumes a great deal of effort, time and cost to automatically generate test cases from extended state chart diagram (b) existing techniques generate a significant number of test cases with less coverage and (c) inefficient test case generation methods for node or path coverage.

This paper aims to resolve the following research issues: (a) minimize size of test cases and test data derived from extended state chart diagram (b) maximize a number of nodes coverage and (c) minimize total time of test case generation from diagrams. This paper proposes an effective method to prepare and generate both of test cases and test data, called "TGfMMD" method. The TGfMMD method is developed to verify the state chart diagram before generation and generate both of test cases and test data from extended state chart diagram. Moreover, this paper proposes to compare to other three test case generation techniques, which are: Dehla's work, Samuels' method and Santiago's technique. As a result, this study found that TGfMMD method is the best to generate the smallest size of test cases with the minimum total time and cover 100% all nodes in the state diagram. Finally, this paper guides the following future works: (a) implement the TGfMMD method in the commercial industry (b) evaluate the proposed method with larger set of states or more complex state chart diagram and (c) improve the ability to verify the state chart diagram in the TGfMMD diagram.

REFRENCES:

[1] Alberto Avritzer and Elaine J. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software", IEEE Transactions on Software Engineering, 21(9):705–716, 1995.

[2] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, Thierry Jeron and Laurent Mounier, "Using UML for Automatic Test Generation", Oxford University Computing Laboratory, Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2000, 2000.

[3] Amaral, "A.S.M.S. Test case generation of systems specified in Statecharts. M.S. thesis –

Laboratory of Computing and Applied Mathematics", INPE, Brazil, 2006.

[4] Annelises A. Andrews, Jeff Offutt and Roger T. Alexander, "Testing Web Applications. Software and Systems Modeling", 2004.

[5] Avik Sinha, Ph.D and Dr. Carol S. Smidts, "Domain Specific Test Case Generation Using Higher Ordered Typed Languages fro Specification", Ph. D. Dissertation, 2005.

[6] Aynur Abdurazik and Jeff Offutt, "Generating Test Cases from UML Specifications", 1999.

[7] A. Bertolino, "Software Testing Research and Practice", 10th International Workshop on Abstract State Machines (ASM'2003), Taormina, Italy, 2003.

[8] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications", Software Engineering Notes, 24(6):146–162, 1999.

[9] A. Jefferson Offutt, Yiwei Xiong and Shaoying Liu, "Criteria for Generating Specification-based Tests", 2007.

[10] A. Jefferson Offutt, Yiwei Xiong and Shaoying Liu, "Criteria for Generating Specification-based Tests", 1999.

[11] A.Z. Javed, P.A. Strooper and G.N. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture", Second International Workshop on Automation of Software Test (AST'07), 2007.

[12] Bogdan Korel, "Automated Software Test Data Generation", IEEE Transaction on Software Engineering, 1990.

[13] Bogdan Korel and Ali M. Al-Yami, "Automated Regression Test Generation", ISSTA98, 1998.

[14] B. Beizer, "Software Testing Techniques", Van Nostrand Reinhold, Inc, New York NY, 2nd edition. ISBN 0-442-20672-0, 1990.

[15] Carl Adam Petri and Wolfgang Reisig, "Petri net", Scholarpedia, 2008.

[16] Cem Kaner, "A Course in Black Box Software Testing", 2004.

[17] Chien-Hung Liu, David C. Kung, Pei Hsia and Chih-Tung Hsu, "Object-Based Data Flow Testing of Web Applications", Proceedings of the First Asia Pacific Conference on Quality Software (APAQS'00), pp. 7-16, Hong Kong, China, 2000.

[18] C. Ramamoorthy, S. Ho, and W. Chen, "On the automated generation of program test data", IEEE Transactions on Software Engineering, SE-2(4):293–300, 1976.

[19] David A. Turner, Arokiya L.M. Joseph, Wonik Choi and Jinseok Chae, "An Activity Oriented Approach for Testing Web Applications", 2008.

[20] David C. Kung, Chien-Hung Liu and Pei Hsia, "An Object-Oriented Web Test Model for Testing Web Applications", In Proceedings of the First Asia Pacific Conference on Quality Software (APAQS'00), page 111, Los Alamitos, CA, 2000.

[21] Dehla Sokenou, "Generating Test Sequences from UML Sequence Diagrams and State Diagrams", 2003.

[22] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification", IEEE Transactions on Software Engineering, 20(5):353-363, 1994.

[23] Flippo Ricca and Paolo Tonella, "Analysis and Testing of Web Applications", Proc. of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada. pp.25-34, 2001.

[24] Harel, D., "Statecharts: a visual formalism for complex system", Science of Computer Programming, v. 8, n., p. 231-274, 1987.

[25] Hassan Reza, Kirk Ogaard and Amarnath Malge, "A Model Based Testing Technique to Test Web Applications Using Statecharts", Fifth International Conference on Information Technology, 2008.

[26] Hetzel, William C., "The Complete Guide to Software Testing", 2nd ed. Publication info: Wellesley, Mass.: QED Information Sciences. ISBN: 0894352423, 1988.

[27] Hung Tran, "Test Generation using Model Checking", Proceeding Conference on Automated Verification, 2001.

[28] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, Inyoung Ko, "Test Cases Generation from UML Activity Diagrams", Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007.

[29] Ian Sommerville, "Software Engineering", 6th edition Section 20, 2000.

[30] Ibrahim K. El-Far and James A. Whittaker, "Model-based Software Testing", 2001.

[31] Jane Huffman Hayes and A. Jefferson Offutt, "Increased Software Reliability through Input Validation Analysis and Testing", 1999.

[32] Jane Huffman Hayes and A. Jefferson Offutt, "Input Validation Testing: A Requirements-Driven, System Level, Early Lifecycle Technique", 2000.

[33] Jeff Offutt, Shaoying Liu, Aynur Abdurazik and Paul Ammann, "Generating Test Data

from State-based Specifications", ISE Department, George Mason University, USA, 2003.

[34] Jim Heumann, "Generating Test Cases From Use Cases", Rational Software, 2001.

[35] Johannes Ryser and Martin Glinz, "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test", 2000.

[36] John E. Bentley, "Software Testing Fundamentals – Concepts, Roles and Terminology", Proceeding with SUGI30, Wachovia Bank, Charlotte NC, 2005.

[37] Mahnaz Shams, Diwakar Krishnamurthy and Behrouz Far, "A Model-Based Approach for Testing the Performance of Web Applications", Proceedings of the Third International Workshop on Software Quality Assurance (SOQUA'06), 2006.

[38] Mani Prasad Kancherla, "Generating Test Templates via Automated Theorem Proving", Technical Report, NASA Ames Research Center, 1997.

[39] Manish Nilawar and Dr. Sergiu Dascalu, "A UML-Based Approach for Testing Web Applications", Master of Science with major in Computer Science, University of Nevada, Reno, 2003.

[40] Mats Grindal, Jeff Offutt and Sten F. Andler, "Combination Testing Strategies: A Survey", 2004.

[41] Mealy, George H., "A Method for Synthesizing Sequential Circuits", Bell Systems Technical Journal, September 1955.

[42] Miao Huaikou and Liu Ling, "A Test Class Framework for Generating Test Cases from Z Specifications", 2000.

[43] Mohammed Benattou, Jean-Michel Bruel and Nabil Hameurlain, "Generating Test Data from OCL Specification", 2002.

[44] Myers, Glenford J., "The art of software testing", Publication info: New York : Wiley. ISBN: 0471043281, 1979.

[45] M. Blackburn and R. Busser, "T-VEC: A tool for developing critical systems", In Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96), pages 237-249, Gaithersburg MD. IEEE Computer Society Press, 1996.

[46] M. Prasanna S.N. Sivanandam R.Venkatesan R.Sundarrajan, "A Survey on Automatic Test Case Generation", Academic Open Internet Journal, 2005.

[47] Nigel Tracey, John Clark, and Keith Mander, "Automated program flaw finding using simulated annealing", In SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), volume 23 of Software Engineering Notes, pages 73–81. ACM Press, 1998.

[48] NIST, "The economic impacts of inadequate infrastructure for software testing", 2002.

[49] Pan, Jiantao, "Software Testing (18-849b Dependable Embedded Systems)", Electrical and Computer Engineering Department, Carnegie Mellon University, 1999.

[50] Percy Antonio, Pari Salas and Bernhard K. Aichernig, "Automatic Test Case Generation for OCL: a Mutation Approach", 2005.

[51] Peter Frohlich and Johannes Link, "Automated Test Case Generation from Dynamic Models", 2000.

[52] Phil McMinn, "Search-based Software Test Data Generation: A Survey", 2004.

[53] Philip Samuel and Anju Teresa Joseph, "Test Sequence Generation from UML Sequence Diagrams", Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2008.

[54] P. Samuel, R. Mall and A.K. Bothra, "Automatic Test Case Generation Using Unified Modeling Language (UML) State Diagrams", IET Software, 2008.

[55] P. Stocks, "Applying Formal Methods to Software Testing", PhD thesis, the Univ. of Queensland, Australia, 1993.

[56] P. Stocks and David Carrington, "A Framework for Specification-Based Testing", IEEE Trans. on Software Engineering, V01.22, No. 11, 1996.

[57] Richard A. DeMillo and A. Jefferson Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transaction on Software Engineering, 1991.

[58] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck, "Test-data generation using genetic algorithms", Software Testing, Verification and Reliability, 9(4):263– 282, 1999.

[59] Sami Beydeda and Volker Gruhn, "BINTEST – binary search-based test case generation", In Computer Software and Applications Conference (COMPSAC), IEEE Computer Society Press, 2003.

[60] Sanjai Rayadurgam and Mats P. E. Heimdahl, "Test-Sequence Generation from Formal Requirement Models", Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering (HASE'01), 2001.

[61] Sanjai Rayadurgam and Mats P.E. Heimdahl, "Coverage Based Test-Case Generation using Model Checkers", 2001.

[62] Sasa Misailovic, Alekasandar Milicevic, Sarfraz Khurshid and Darko Marinov, "Generating Test Inputs for Fault-Tree Analyzers using ImperativePredicates", Proceeding in Workshop on Advances and Innovations in Systems Testing, 2007.

[63] Sara Sprenkle, Emily Gibson, Sreedevi Sampath and Lori Pollock, "A Case Study of Automatically Creating Test Suites from Web Application Field Data", TAV-WEB'06, 2006.

[64] Stefania Gnesi, Diego Latella and Mieke Massink, "Formal Test-case Generation for UML Statecharts", Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Enginerring Age, 2004.

[65] Suet Chun Lee and Jeff Offutt, "Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis", 2001.

[66] S.J. Cunning and J.W. Rozenblit, "Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems", 1999.

[67] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton and B.M. Horowitz, "Model-Based Testing in Practice", Proceeding of ICSE'99, 1999.

[68] U. Farooq, C.P. Lam and H. Li, "Towards Automated Test Sequence Generation", 19th Australian Conference on Software Engineering, 2008.

[69] Valdivino Santiago, Ana Silvia Martins do Amaral, N.L. Vijaykumar, Maria de Fatima, Mattiello-Francisco, Eliane Martins and Odnei Cuesta Lopes, "A Practical Approach for Automated Test Case Generation using Statecharts", 2006.

[70] Vijaykumar, N. L.; Carvalho, S. V.; Abdurahiman, V., "On proposing Statecharts to specify performance models", International Transactions in Operational Research, 9, 321-336, 2002.

[71] Wagner, F., "Modeling Software with Finite State Machines: A Practical Approach", Auerbach Publications, 2006.

[72] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong and Zheng Guoliang, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method", Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004.

[73] W. Eric Wong, Yu Lei and Xiao Ma, "Effective Generation of Test Sequences for Structural Testing of Concurrent Programs", Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), 2005.

[74] W.T. Tsai, X. Wei, Y. Chen, R. Paul and B. Xiao, "Swiss Cheese Test Case Generation for Web Services Testing", IEICE Transactions (IEICET) 88-D(12):2691-2698, 2005.

[75] Xiaoping Jia and Hongming Liu, "Rigorous and Automatic Testing of Web Applications", 2002.

[76] Xiaoping Jia, Hongming Liu and Lizhang Qin, "Formal Structured Specification for Web Application Testing", Proc. of the 2003 Midwest Software Engineering Conference (MSEC'03), Chicago, IL, USA, pp.88-97, 2003.