# Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 4 (4): 265-287, 2010
ISSN 1819-4311


# A Test Case Generation Process and Technique

Nicha Kosindrdecha and Jirapun Daengdej
Autonomous System Research Laboratory,
Faculty of Science and Technology, Assumption University, Thailand

**Abstract:** This study aims to improve an automated test case generation method to minimize a number of test cases while maximizing an ability to identify critical domain specific requirements. It has been proven that the software testing phase is one of the most critical and important phases in the software development life cycle. In general, the software testing phase takes around 40-70% of the effort, time and cost. This area has been well researched over a long period of time. Unfortunately, while many researchers have found methods of reducing time and cost during the testing process, there are still a number of important related issues that need to be researched. This study introduces a new test case generation process with a requirement prioritization method to resolve the following research problems: (1) inefficient test case generation techniques with limited resources (2) lack of an ability to identify critical domain requirements in the test case generation process (3) inefficient automated test case generation techniques and (4) ignoring a number of generated test cases. In brief, the contributions are to: (1) study a comprehensive set of test case generation techniques since 1990, (2) compare existing test case generation methods and address the limitations of each technique, (3) introduce a new classification of test case generation techniques, (4) define a new process to generate test cases by proposing a requirement prioritization method and (5) propose a new effective test generation method.

**Key words:** Test case generation, testing research issues, test generation, test generation method, requirement prioritization and test generation process

## INTRODUCTION

According to the waterfall software development life cycle (SDLC) below, basically there are five phases in the cycle, which are: (1) requirements, (2) design, (3) implementation (also known as development), (4) verification (also known as software testing) and (5) maintenance. Software testing phase is the process of executing a program or system with the intent of finding errors (Myers, 1979). It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results (William, 1988). 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.

**Corresponding Author:** Nicha Kosindrdecha, Autonomous System Research Laboratory,
Faculty of Science and Technology, Assumption University, Thailand
Tel: 66023773687, 66898116163

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 (Beizer, 1990) and a recent detailed survey in the United States (NIST, 2002) quantified the high economic impacts of an inadequate software testing infrastructure.

In addition, Bentley (2005) stated 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.

It is concluded from this that the impact of inadequate testing can be root-cause problems of: (1) increased failures due to poor quality (2) increased software development costs (3) increased time to market due to inefficient testing and (4) increased market transaction costs (NIST, 2002).

In software testing, Ian (Sommerville, 2000) stated that there are four processes, which are: (1) design test cases (also known as test case generation process), (2) prepare test data, (3) run program with test data and (4) compare results to test cases. The test case generation process is a fundamental and the most critical process in the software testing process (Sommerville, 2000; Bertolino, 2003; Prasanna *et al*., 2006). Bertolino (2003) stated 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 (Antonio *et al*., 2005; Offutt *et al*., 1999; Heumann, 2001; Turner *et al*., 2008). In addition, Kaner (2003) 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. Also, they showed that the test case generation process is an extensively researched activity and consumes a lot of time and cost. Therefore, many researchers from 1990 to 2006 mentioned that automated test case generation is one approach to reducing time and cost during the test case generation process. Many methods have been proposed to identify a set of test cases, such as Sanjai's work (Rayadurgam and Heimdahl, 2001a), Hyungchoul's work (Kim *et al*., 2007; Chen *et al*., 2008; Frohlich and Link, 2000).

This study reviews test case generation methods researched since 1990, such as random approaches, goal-oriented techniques, specification-based techniques, sketch diagram based techniques and source code based techniques. Also, this study shows that the outstanding research issues that motivated this study are: (1) existing test case generation techniques assume explicitly that there are unlimited resources available during the test case generation process, (2) existing methods lack the ability to identify and reserve the critical domain requirements in the test case generation process and (3) not all existing techniques concentrate on generating a minimal set of test cases with the maximal ability to reveal faults.

This study introduces a new test case generation process based on all existing test case generation techniques. Also, this paper proposes a test case generation method to address the above three research issues.

## LITERATURE REVIEW

This section surveys and describes the software testing process, a test case generation process and all recent research of test case generation techniques. The following paragraphs describe the general process of running software testing activities. This study includes the software testing process provided by Ian (Sommerville, 2000), who is the author of well-known software testing books, as follows:

- **Design test cases:** The purpose of this step is to generate and prepare a set of test cases. Therefore, the outcome of this step is a set of test cases. A set of test cases may be represented in Excel format, as Word documents or as a database
- **Prepare test data:** The purpose of this step is to generate and prepare test data for each test case. The outcome of this step is a set of test data
- **Run program with test data:** This is an execution test step. Test case and test data will be run in this step. The result of this step is actual system output
- **Compare results to test cases:** This step is used to compare the system output to expected output in the test case. The milestone of this step is a test report of running the test case with test data

Sommerville (2000), the test case generation process (or the process of designing test cases) is the first and the most important process in software testing. The test case generation process is also known as a "test development" process in Pan's work (Pan, 1999). The test case generation process has always been fundamental to the testing process. Bertolino (2003) 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 (Prasanna *et al*., 2005) such as random approaches, goal-oriented technique, 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 (Jia *et al*., 2003; Ricca and Tonella, 2001; Wu and Offutt, 2002; Wu *et al*., 2004).

Random techniques determine a set of test cases based on assumptions concerning fault distribution. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken. Specification-based techniques design a set of test cases from formal requirement specifications. Generating test cases of complex software from non-formal specification can result in incorrect implementations leading to the necessity to test them for conformance to its specification (Santiago *et al*., 2006). Sketch diagram based techniques derive test cases from UML diagrams. The UML diagrammatic technique is the most widely used in the software design phase. Many diagrams are used in generating a set of test cases, such as use case diagram, activity diagram and state chart diagram. Source code-based techniques (or Path-oriented techniques) generally use a control flow graph to identify paths to be covered and generate appropriate test cases for those paths.

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

**Specification-based Test Case Generation Techniques**

Specification-based techniques are methods to generate a set of test cases from specification documents such as a formal requirements specification (Cunning and Rozenblit, 1999; Tran, 2001; Rayadurgam and Heimdahl, 2001a; Nilsson *et al*., 2006; Tsai *et al*., 2005), Z-specification (Huaikou and Ling, 2000; Jia and Liu, 2002; Jia *et al*., 2003) and Object Constraint Language (OCL) specification (Antonio *et al*., 2006).

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 (Subraya and Subrahmanya, 2000).

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 (Memon *et al*., 1999). 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 (Offutt *et al*., 1999).

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: (1) 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 (Liu *et al*., 2001) and (2) 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 (Cunning and Rozenblit, 1999; Kancherla, 1997), model checkers (Tran, 2001; Rayadurgam and Heimdahl, 2001a; Nilsson *et al*., 2006) and hierarchy approaches (Huaikou and Ling, 2000; Jia and Liu, 2002; Jia *et al*., 2003). The following paragraphs describe existing specification-based techniques that have been proposed since 1997.

Antonio *et al*. (2005) 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.

Offutt *et al*. (1999) 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 (Blackburn and Busser, 1996). It is used to derive disjunctive normal form constraints, which are solved to generate tests. Also, their approach is related to Weyuker *et al*. (1994) 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.

Kancherla (1997) used a form of specification-based testing that employs the use of an automated theorem prover to generate test cases. A similar approach was developed using a model checker on stat-intensive systems. The method applies to systems with functional rather than stat-based behaviors. The approach allows for the use of incomplete specifications to aid in generation of tests for potential failure cases. He suggested a new method of testing software based on the formal specification. He used the Prototype Verification System (PVS) and its in-built theorem prover to derive test cases corresponding to the properties stated in the requirements.

Cunning and Rozenblit (1999) were 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 study, 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 (Cunning and Rozenblit, 1999).

Tran (2001) 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.

Rayadurgam and Heimdahl (2001b) presented a method for automatically generating test cases to structural coverage criteria. They 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. They provided a formal framework that is: (1) suitable for defining their test-case generation approach and (2) easily used to capture finite state representations of software artifacts such as program code, software specifications and requirements models. They 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, they 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 (Gargantini and Heitmeyer, 1999), 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.

Nilsson *et al.* (2006) has proposed a model based method for generating test cases to test timeliness by using heuristic driven simulation. Their approach is perfectly suited to generating test cases for small real-time systems that contain shared resources, precedence constraints and few sporadic tasks. Conversely, in dynamic real-time systems there are many sporadic tasks, making model-checking impractical. For these dynamic real-time systems, they proposed an approach where a simulation of each mutant model is iteratively run and evaluated using genetic algorithms with application specific heuristics. By using a simulation-based method instead of model-checking for execution order analysis, the combinatorial explosion of full state exploration is avoided. Furthermore, they conjectured that it is easier to modify a system simulation than a model-checker, to correspond to the architecture of the system under test. In their paper, they focused on genetic algorithms.

They included three types of functions needed to solve the specific search problem. Those three functions are: (1) a genome mapping function, (2) heuristic cross-over functions and (3) fitness function.

**Sketch Diagram-based Test Case Generation Techniques**

Sketch diagram-based techniques are methods to generate test cases from model diagrams like UML Use Case diagram (Heumann, 2001; Ryser and Glinz, 2000; Nilawar and Dascalu, 2003), UML Sequence diagrams (Javed *et al*., 2007) and UML State diagrams (Sinha and Smidts, 2005; Santiago *et al*., 2006; El-Far and Whittaker, 2001; Cavarra *et al*., 2000; Reza *et al*., 2008; Kung *et al*., 2000; Shams *et al*., 2006; Andrews *et al*., 2004). 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 VandV 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 (Javed *et al*., 2007). The following paragraphs describe existing specification-based techniques that have been proposed since 2000.

Heumann (2001) 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: (1) for each use case, generate a full set of use-case scenarios (2) for each scenario, identify at least one test case and the conditions that will make it execute and (3) for each test case, identify the data values with which to test.

Ryser and Glinz (2000) raised the practical problems in software testing as follows: (1) lack of planning/time and cost pressure, (2) lack of test documentation, (3) lack of tool support, (4) formal language/specific testing languages required, (5) lack of measures, measurements and data to quantify testing and evaluate test quality and (6) 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: (1) preliminary test case and test preparation during scenario creation, (2) test case generation from Statechart and dependency charts and (3) test set refinement by application dependent strategies (intuitive, experience-based testing).

Nilawar and Dascalu (2003) 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 J. Heumann's four-steps (Heumann, 2001), as follows: (1) prioritize use cases based on the requirement traceability matrix, (2) generate tentatively sufficient use cases and test scenarios, (3) for each scenario, identify at least one test case

and the conditions and (4) 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.

Sinha and Smidts (2005) 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.

Santiago *et al.* (2006) focused on test sequence generation from a specification of a reactive system, space application software, in Statecharts (Harel, 1987) and the use of PerformCharts (Vijaykumar *et al.*, 2002). 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 (Amaral, 2006). 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: (1) a dual graph is created from the original one, by converting arcs into nodesm (2) 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, (3) the dual graph is transformed into a "Eulerized" graph by balancing the polarity of the nodes and (4) finally, the nodes are traversed registering those that are visited.

El-Far and Whittaker (2001) were interested in model-based testing and generating test cases from finite state machines. The difficulty of generating test cases from a model depends on the nature of the model. Models that are useful for testing usually possess properties that make test generation effortless. Sometimes generation processes can be automated. For some models, one must go through combinations of conditions described in the model. In the case of finite state machines, it is as simple as implementing an algorithm that randomly traverses the state transition diagram. The sequences of arc labels along the generated paths are, by definition, tests.

Cavarra *et al.* (2000) described a modeling architecture for the purposes of model based verification and testing. Their architecture contains two components. The first component of the architecture is the system model, written in UML; this is a collection of class, state and object diagrams: the class diagram identifies the entities in the system; the state diagrams explain how these entities may evolve; the object diagram specifies an initial configuration. The second component, again written in UML, is the test directive; this consists of particular object and state diagrams: the object diagrams are used to express test constraints and coverage criteria; the state diagrams specify test purposes. The system model and the test directives can be constructed using any of the standard toolsets, like Rational Rose.

Reza *et al.* (2008) discussed a model-based testing method for web applications that utilizes behavioral models of the software under the test (SUT) from Statechart models originally devised by Harel (1987, 1988). Statechart models can be used both for modeling and generating test cases for a web application. The main focus of their work is on the front end design and testing of a web application. As such, they utilize the syntax of the web pages to guide the specification of the Statecharts. Their approach is a systematic way to test the front-end functionality of a web application. For the most parts, they are concerned with verifying that the links, forms and images in the web application under test function according to the specification documents. Furthermore, they address how to model the

web application with Statechart diagrams in their work. To generate test cases from Statechart diagram, they defined 5 test coverage criteria: (1) all-blobs, (2) all-transitions, (3) all-transition-pairs, (4) all-conditions and (5) all-paths.

**Source Code-based Test Case Generation 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: (1) test case ID, (2) test data, (3) test sequence (also known as test steps), (4) expected result, (5) actual result and (6) pass/fail status. The following paragraphs describe the source code-based techniques that have been proposed since 1999.

Beydeda and Gruhn (2003) presented a novel approach to automated test case generation. Several approaches have been proposed for test case generation, mainly random, source code-based (or path-oriented), goal-oriented and intelligent approaches (Pargas *et al*., 1999). Random techniques determine test cases based on assumptions concerning fault distribution, e.g., (Avritzer and Weyuker, 1995). 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., (Ramamoorthy *et al*., 1976), whereas dynamic techniques obtain the necessary data by executing the program under test e.g., (Korel, 1990). Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken e.g., (Pargas *et al*., 1999). Intelligent techniques of automated test case generation rely on complex computations to identify test cases e.g., (Gupta *et al*., 1998). Another classification of automated test case generation techniques can be found in (Gupta *et al*., 1998). Their algorithm proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that (Korel, 1990). The path to be covered is considered step-by-step, i.e., the goal of covering a path is divided into sub-goals, test cases are then searched to fulfill them. The search process, however, differs substantially. In Bogdan's work (Korel, 1990), 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. Turner *et al*. (2008) 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.

Yang *et al*. (1999) presented web application architecture to support testing of the web application. The architecture covers application model extraction, test execution automation and test design automation. In addition, practitioners normally use a graph-based application

model to represent the behavior of web-based applications. They are interested in extending the control flow graph (e.g., nodes, branches and edges) to model web applications. The nodes in the control flow graph represent a programming module (e.g., single file such as .html, .cgi and .asp). The branch could be the user branch and application branch. The user branch represents the user selecting one of the hyperlinks from the browsed document in the browser. The application branch represents the current programming module forwarding control to other programming modules for further processing based on application logic. The extended model is further used to generate test cases by applying the traditional flow-based test cases generation technique. They adopt two path testing strategies: statement and branch coverage for their environment. The IEEE software testing standard regards statement coverage as the minimum testing requirement. Real world, practical program testing requires both the statement and branch coverage. They declared four major steps for their testing activities in their framework: (1) application model construction, (2) test case construction and composition, (3) test case execution and (4) test result validation and measurement.

In the conclusion, the section shows that there are three major sources used for software test engineers to design and generate test cases, which are: (1) formal requirement specifications, (2) sketch diagrams, like use case diagrams, activity diagrams and state chart diagrams and (3) control flow graphs derived from the source code. Additionally, this paper shows that many researchers proposed specification-based techniques and sketch diagram based techniques. A few researchers concentrate on source code based techniques. All existing test case generation techniques have advantages and limitations. With regard to using the test case generation techniques, there are a significant number of issues that need to be addressed in next section.

## RESEARCH CHALLENGES

This section discusses the details of research issues related to test case generation techniques and research problems which motivated this study. Every test case generation technique has weak and strong points, as addressed in the literature survey. In general, referring to the literature review, the following lists major outstanding research challenges.

- **Inefficient Test Case Generation Techniques with Limited Resources (e.g., Time, Effort and Cost):** The software testing phase of a project is often awarded lowest priority. Typically, software testing engineers have a small amount of time, effort and cost to plan and design test case, run test cases and evaluate test cases respectively. Existing techniques are not effective for complex applications with limited resources (e.g., time, effort and cost), both traditional and web applications. An example of a complex web application is an application with dynamic behavior, heterogeneous representations, or novel control and data flow mechanisms
- **Lack of Ability to Identify Critical Domain Requirements:** The existing test case generation techniques lack the capability to address domain specific requirements, because those requirements are not explicitly discussed in the specification document. For an example of this problem (Nilsson *et al*., 2006), where a technique is proposed to generate test cases for real-time systems
- **Ignore Size of Test Case:** Existing test case generation techniques aim to generate test cases which maximize cover for each scenario. Sometimes, they generate very large test cases which are impossible to execute given limited time and resources

## PROPOSED METHODS

This section introduces a new 2D-4A-4D process to design and prepare test cases. Also, this section discusses a proposed method that resolves the above research problems. The proposed method aims to: (1) prioritize a huge set of requirements in order to improve the effectiveness of test case generation techniques while there are limited resources, (2) increase the ability to cover more critical domain requirements during the generation process and (3) minimize and generate a small set of test cases with high ability to reveal faults.

### Test Case Generation Process

This section presents a new 2D-5A-4D process to generate a set of test cases introduced by using the above comprehensive literature review and previous works (Kosindrdecha and Daengdej, 2009).

Figure 1 describes an overview of existing test case generation process. The proposed process shows that test case generation methods typically generate test cases from the following sources: (1) requirement specification document, (2) UML diagram and (3) source code. The following describes the process in details.

There are two processes in the test case generation technique, which break down briefly as follows:

### Define

This is a first process that allows software testing engineers to gather, analyze and define all pre-requisite and required information, such as requirements, constraints and type of testing. There are four sub-processes described shortly as follows:

Table 1 describes the first proposed process in details. It contains five columns: sub-process, purpose, description, input and output. The sub-process is a sequential process to analyze requirements before generating test cases. The purpose is a goal that each process aims to achieve. The description describes a short summary of what the process is and means to software test engineers. The input is a required pre-requisite for each process while the output is an outcome of each process.

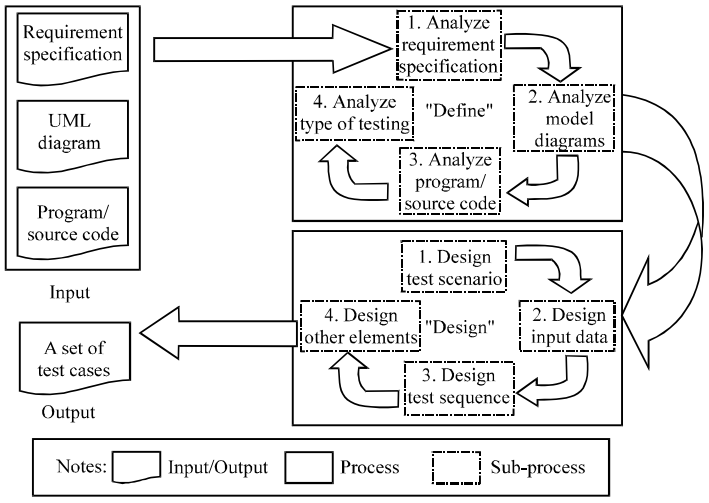


Fig. 1: 2D-4A-4D process to generate test cases

Table 1: The first process in "2D-4A-4D" test case generation process

| Sub-process | Purpose | Description | Input | Output |
|---|---|---|---|---|
| Analyze requirements specification | • To be able to perform black-box testing activities<br>• To understand requirements or function specification document<br>• To verify and validate between the requirements and system | Software test engineers need to walk through and understand all requirements or function in the specification | Requirement or Function Specification Document | Understanding of requirements, constraints and an overview of how to test in general |
| Analyze model diagrams | • To be able to perform black-box testing activities<br>• To get better understand the design diagrams<br>• To verify that the behavior of system is match to the design | Software test engineers have to analyze the detail design diagrams, such as UML Use Case diagram, UML Activity diagram and State Chart diagram | Detailed design diagrams | Understanding of information in the diagrams in order to be able to derive tests from them |
| Analyze program/ source code | • To be able to perform white-box testing activities<br>• To be able to understand and help software developer to test program/source code | Software test engineers have to analyze and walk through program/source code in order to run a white-box testing activities | Available of program or source code | Understanding of the testing strategy/approach for which or how many line of code in the program should be tested In addition, another output should be a control flow graph transformed from source code |
| Analyze type of testing | • To be able to identify which type of testing should be executed<br>• To allow to design test strategy or plan for each testing type (ie, functionality, performance and security) | Software test engineers need to analyze and identify which type of testing should be executed | Requirement Specification and Diagrams | Understanding a type of testing in order to prepare a proper testing strategy or plan |

Table 2: The second process in 2D-4A-4D test case generation process

| Sub-Process | Purpose | Description | Input | Output |
|---|---|---|---|---|
| Design test scenario | • To design a high level scenario for testing<br>• To be able to use as a reference to verify the requirements and testing scenario | Software test engineers have to design many testing scenarios to cover all requirements or function | Requirement Specification, Diagrams and Source Code | A set of test scenarios |
| Design input data | • To design a set of input data used during a test execution phase<br>• To design a realistic input data, both of positive and negative data<br>• To design a special case of input data (e.g. special characters or special combination of symbols and characters) | Software test engineers have to design many sets of input data that are used for testing | • Requirement Specification and Source Code<br>• A set of test scenarios | Many sets of input data |
| Design test sequence | • To design a sequence of testing activities<br>• To understand test steps of each test scenario | Software testing engineers have to design a set of test sequence or steps for each test scenarios | • Requirement Specification and Detailed Diagrams<br>• A set of test scenarios | Many set of test sequences |
| Design other elements | To complete designing a set of test cases | Software test engineers must complete a set of test cases by adding additional required elements, such as actual results, dependencies, business impact and defect id | A set of test scenarios | A complete set of test case |

## Design

This is a second process that aims to design, prepare and generate all elements in a set of tests, such as test data, test sequence and dependencies of each test case. This process contains the following sub-processes:

Table 2 describes the second proposed process in details. It also contains five columns: sub-process, purpose, description, input and output. The sub-process is a sequential process to prepare and generate all test elements, such as test scenario, test sequence and test data. The purpose is a goal that each process aims to achieve. The description describes a short summary of what the process is and means to software test engineers. The input is a required pre-requisite for generating test cases while the output is a testing artifact.

The above process can help software test engineers to design, prepare and generate all elements in a set of test cases. It can ensure that all elements are well-prepared. In addition,

this process contains all required important or critical elements that can be used in the general commercial industry, such as test scenario, test case, test data, test sequence and dependencies of each test case.

**Test Case Generation Technique**

Many researchers mentioned that prioritizing requirements is one of the most important activities during the software testing process, particularly in large complex projects, for instance Karl's work (Wiegers, 1999), Donald's study (Firesmith, 2004) and Nancy's recent work (Mead, 2008).

Karl confirmed that software testing engineers (or even developers) do not always know which requirements are the most important to the customers. Not only this, but customers also cannot judge the cost and technical difficulty associated with the requirements. In fact, most of projects have limited resources such as time, human resources and cost (Firesmith, 2004). It is difficult to meet the customer's expectation with limited resources. Firesmith (2004) has proposed the requirement prioritization techniques in order to prioritize and schedule requirements which are most important. Also, there are many benefits of prioritizing requirements such as improved customer satisfaction and a greater ability to address all critical requirements and to prioritize investments.

Mead (2008) recommended the requirements prioritization process, because it is an important activity. There is a recommendation that all stakeholders select candidate prioritization techniques, develop selection criteria to pick one and apply it to decide which security requirements to implement when. During the prioritization process, the stakeholders can verify that everyone has the same understanding about the requirements and further examine any ambiguous requirements. After everyone reaches consensus, the results of the prioritization exercise will be more reliable.

The above literature review shows that existing test case generation techniques derive test cases directly from requirements, specification requirement documents or diagrams. None of them are concerned with prioritizing a huge set of requirements. Practically, there are a huge set of requirements in the software development, particularly in large complex software. Thus, prioritizing requirements before preparing and generating test cases is one of the most important activities, which software test engineers can not ignore.

The following proposes a new process to generate a set of test cases, by adding an additional process, called 2D-5A-4D. It is included the requirement prioritization process, as follows:

Figure 2 introduces a new test case generation process. This study proposes to insert an additional process in order to maximize critical domain specific requirements while minimizing a number of test cases during testing process. The requirement prioritization process contains primarily two major processes: (1) requirement classification by MoSCoW method and (2) requirement prioritization by using cost-valued approach.

The following describes a process flow in details, during the requirement prioritization process:

From the Fig. 3, the steps can be shortly described as follows:

Check whether requirements are critical domain requirements or not by using MoSCoW method (Tierstein, 1997). If they are not critical, then flag those requirements as low priority. This is because the current test case generation techniques may ignore the critical domain requirements (Sinha and Smidts, 2005; Nilsson *et al*., 2006).

If requirements are critical domain requirements, then classify those requirements according to the types identified in the literature survey, which are: Functionality, performance and security requirements.

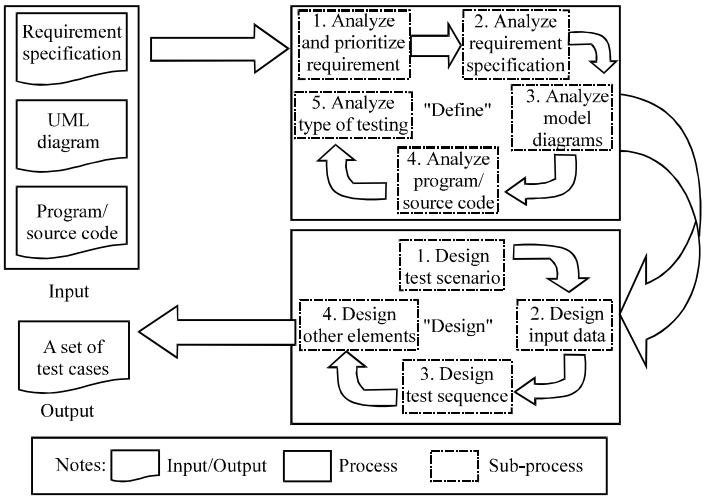


Fig. 2: A New 2D-5A-4D test case generation process

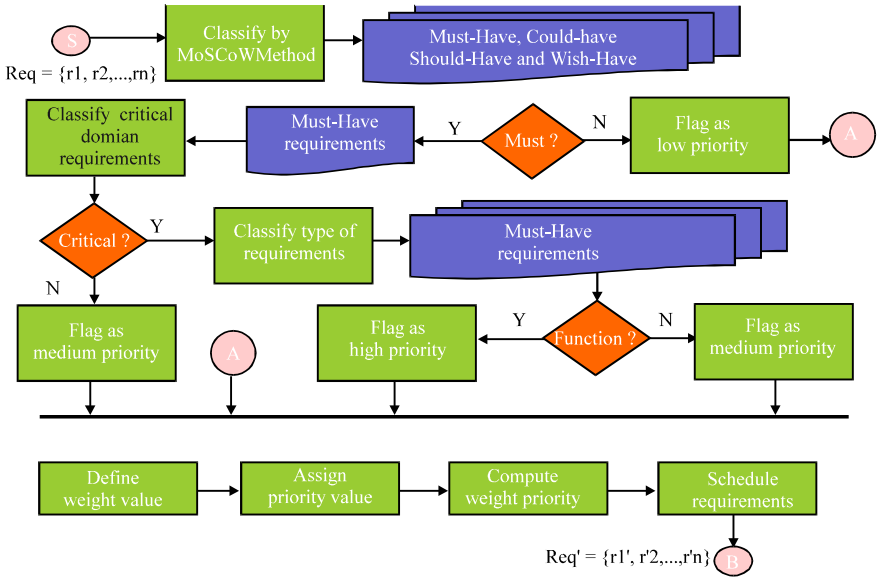


Fig. 3: Flow chart of proposed requirement prioritization method

The functionality requirements are assigned high priority whereas other types of requirements can be assigned medium. This is because many researchers have proven that functionality testing is one of the most important topics in software testing.

Afterward, compute the implementation cost using a cost-value approach. To implement each requirement, developing and testing phases are required. Therefore, the following formula has been proposed to compute the total cost for each requirement:

$$TC = (WValue1*CostImp) + (Wvalue2*CostTest)$$

Where:

TC : The total cost for each requirement
Wvalue1 : A weight value that is assigned for cost of implementation
CostImp : The total cost of implementation, such as analysis, design and develop
Wvalue2 : A weight value that is assigned for cost of implementation
Cost test : The total cost of testing, including planning, designing and evaluating

Apply the concept of the numeral assignment technique by assigning each requirement a number on a scale of 1 to 5 to indicate its importance.

Calculate weight value for each requirement by the following formula:

$$WV = (TC)*(Imp)$$

Where:

WV : A weight value
TC : The total cost mentioned in step 4
Imp : An importance indicator stated in step 5

Prioritize requirements by weight value.

## EVALUATION

This section describes an experiments design, measurements and results.

### Experiments Design

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

- **Prepare Experiment Data:** Before evaluating the proposed methods and other methods, experiment data must be prepared. In this step, 50 requirements and 50 use case scenarios are randomly generated
- **Generate Test Scenarios and Test Cases:** A comparative evaluation has been carried out between the proposed test scenario algorithms, Heumann's technique (Heumann, 2001), Ryser's method (Ryser and Glinz, 2000), Nilawar's algorithm (Nilawar and Dascalu, 2003) and the proposed method, called 2D-5A-4D, presented in the previous section. It includes a prioritization requirement algorithm prior to generating the set of test scenarios and test cases
- **Evaluate Results:** In this step, the comparative generation methods are executed by using 50 requirements and 50 use case scenarios. These methods are executed 10 times to find the average percentage of critical domain requirement coverage, the size of test cases and total generation time. In total, there are 500 requirements and 500 use case scenarios executed in this experiment

### Measurement Metrics

The section lists the measurement metrics used in the experiment. This paper proposes to use three metrics, which are: (1) size of test cases, (2) total time and (3) percentage of critical domain requirement coverage. The following paragraphs describe details of three proposed metrics.

**Size of Test Cases**

This is the total number of generated test cases, expressed as a percentage, as follows:

$$\% \text{ Size} = (\# \text{ Size} / \# \text{ of Total Size})*100$$

Where:

- % Size is the percentage of the number of test cases
- # of Size is the number of test cases that each method generates
- # of total size is the maximum number of test cases in the experiment, which is assigned 1,000

**Total Time**

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

$$\text{Total} = \text{Preparation time} + \text{Compile time} + \text{Running time}$$

Where:

- Total is the total amount of times consumed by running generation methods
- Preparation time is the total amount of time consumed by preparation before generating test cases
- Compile time is the time to compile source code/binary code in order to execute the program
- Running time is the total time to run the program under this experiment

**Percentage of Critical Requirement Coverage**

This is an indicator to identify the number of requirements covered in the system, particularly critical requirements and critical domain requirements (Sinha and Smidts, 2005). Due to the fact that one of the goals of software testing is to verify and validate requirements covered by the system, this metric is a must. Therefore, a high percentage of critical requirement coverage is desirable. It can be calculated using the following formula:

$$\% \text{ CRC} = (\# \text{ of Critical}/\# \text{ of Total})*100$$

Where:

- % CRC is the percentage of critical requirement coverage
- # of Critical is the number of critical requirements covered
- # of Total is the total number of requirements

## RESULTS AND DISCUSSION

This section shows an evaluation of the results of the above experiment. This section presents a graph that compares the above proposed method to the other three existing test

case generation techniques, based on the following measurements: (1) size of test cases, (2) critical domain coverage and (3) total time. Those three techniques are: (1) Heumman's method, (2) Ryser's work and (3) Nilawar's approach. There are two dimensions in the following graph: (1) horizontal and (2) vertical axis. The horizontal represents three measurements whereas the vertical axis represents the percentage value.

Figure 4 showed an evaluation of results and compares a number of test cases, critical domain specific requirement coverage and total generation time. The above graph showed that the above proposed method generates the smallest set of test cases, at 80.80% whereas the other techniques exceed 97%. Those techniques generated a larger set of test cases than the set generated by the proposed method. The literature review revealed that the smaller set of test cases is desirable. Also, the graph showed that the proposed method consumes the least total time during a generation process. It used only 30.20%, which was slightly less than the others. Finally, the graph presented that the proposed method scores best in critical domain coverage. Its percentage was much greater than other techniques' percentage, over 30%. The following table ranked test case generation techniques used in the experiments, based on the above measurements, 1 being the first and 4 the last.

Table 3 shows a ranking of each comparative test case generation method. In the table, it is concluded that our proposed method is the most recommended method to minimize number of tests and generation time while maximizing coverage of critical domain specific requirements.

This section discusses the above evaluation result. Our experiment found that our proposed method is the most recommended test case generation technique to minimize a number of test cases. Also, our experiment showed that our method is the best method

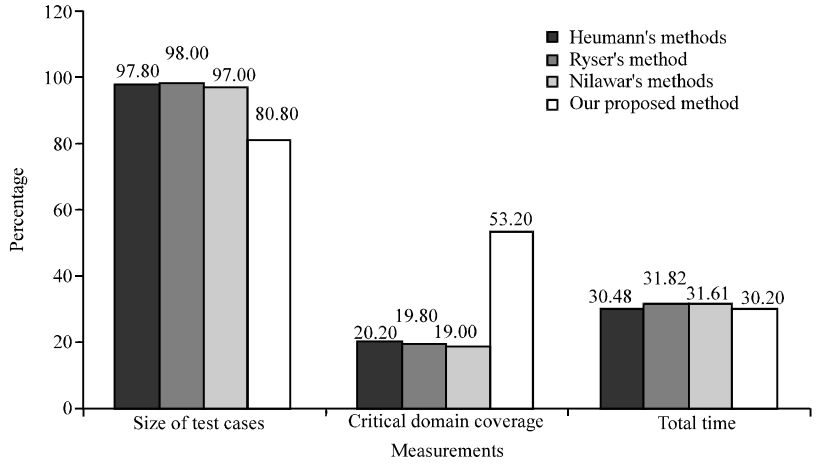


Fig. 4: Evaluation of results of test case generation methods

Table 3: Test case generation techniques ranking table

| Methods | Size of test cases | Critical domain coverage | Total time |
|---|---|---|---|
| Heumman's method | 3 | 2 | 2 |
| Ryser's method | 4 | 3 | 4 |
| Nilawar's method | 2 | 4 | 3 |
| The proposed method | 1 | 1 | 1 |

comparing to other methods, like Heumann, Ryser and Nilawar. Obviously, those methods generate larger number of test cases. The following shows a comparison result in term of numbers of test cases:

Figure 5 compares four test case generation techniques in term of numbers of test cases. The horizon axis represents a number of test cases. The proposed method is by far better than other three methods. Generally, test case generation methods with the smallest number of test cases are desirable.

The following represents a comparison between a number of test cases and coverage of critical domain specific requirements. The horizon axis presents as a number of test cases while the vertical gives domain specific requirement coverage.

Figure 6 shows that our proposed method generate and minimize a number of test cases while preserving a high ability to cover domain specific requirements. Also, it is shows that our method is by far better than other existing test case generation methods in term of a number of tests and coverage.

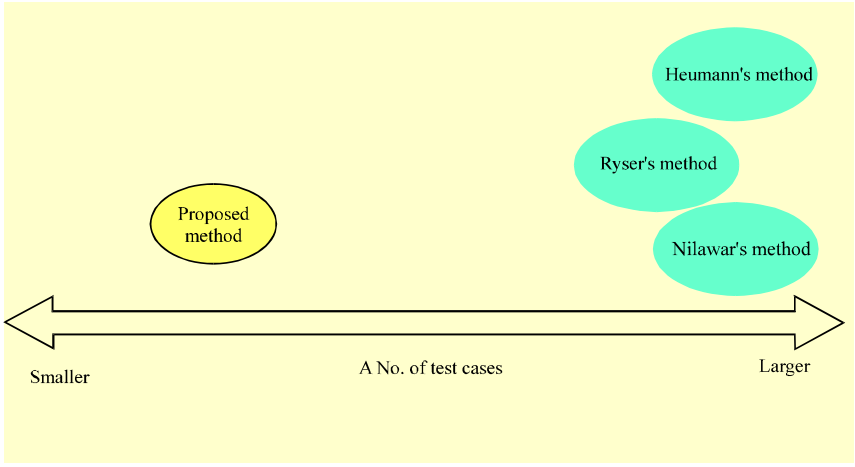


Fig. 5: A comparison result in term of numbers of test cases

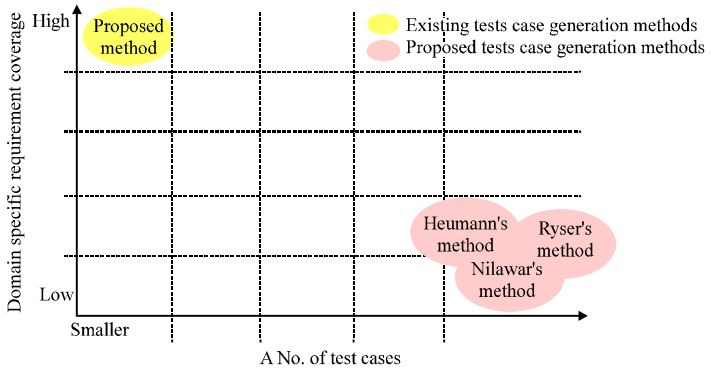


Fig. 6: A comparison between a number of tests and domain requirement coverage

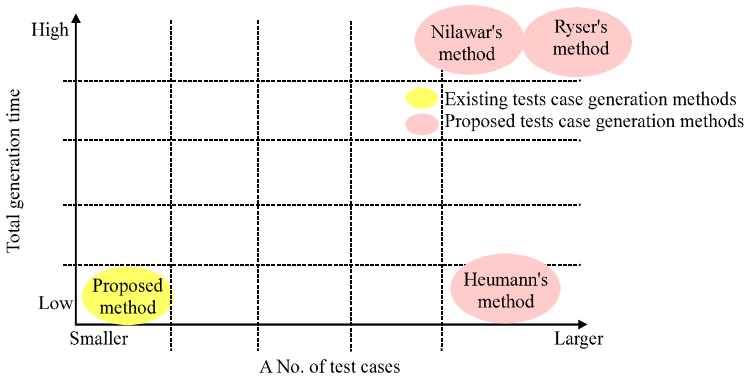


Fig. 7: A comparison between a number of tests and total generation time

The following displays and compares a number of test cases and a total generation time among four test case generation techniques. The horizon axis represents a number of test cases. The vertical presents the total time.

Figure 7 shows that our proposed method is the most recommended methods for the smallest number of test cases and the lowest total generation time. We found that Heumann's method is also the most recommended method for the lowest total time, but unfortunately it generates a larger number of test cases comparing to our method.

## CONCLUSIONS AND FUTURE WORK

This study proposes a new test case generation process, called "2D-4A-4D". The new procedure contains two main processes: (1) definition and (2) design. The first process is composed of four sub-processes, called 4A, which are: (1) analyze requirement specification, (2) analyze design diagrams, (3) analyze source code and (4) analyze type of testing. The second process is also composed of four sub-processes, called 4D, which are: (1) design test scenario, (2) design input data (3) design test sequence and (4) design other elements in the set of test case. There are many research challenges and gaps in the test case generation area. However, this study concentrates on resolving the following research problems: (1) an inefficient test case generation method with limited resources, (2) inability to identify and cover critical domain requirements and (3) an ignorance of a size of test cases. This paper proposes an effective test case generation process by adding an additional prioritization process into the "2D-4A-4D" process. The new process aims to improve the ability to: (1) generate test cases with limited resources, (2) include more critical domain requirements and (3) minimize the size of test cases. The new process is called "2D-5A-4D". This study proposes to compare to other three test case generation techniques, which are: Heummann's work, Ryser's method and Nilawar's technique. As a result, this study found that the proposed method performs best at generating the smallest size of test cases with maximum critical domain coverage and the least time consumed in the test case generation process. The future work is to evaluate with large scale of data and commercial systems.

## REFERENCES

Amaral, 2006. A.S.M.S. Test case generation of systems specified in Statecharts. M.S. Thesis, Laboratory of Computing and Applied Mathematics, INPE, Brazil.

Andrews, A.A., J. Offutt and R.T. Alexander, 2004. Testing web applications. Software and Systems Modeling, http://cs.gmu.edu/~offutt/classes/821-webtest/papers/webtest-821.pdf.

Antonio, P., P. Salas and B.K. Aichernig, 2006. Automatic test case generation for OCL: A mutation approach, Proceedings of 5th International Conference Quality Software, January 2006, IEEE Computer Society, pp: 64-71.

Avritzer, A. and E.J. Weyuker, 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Transactions Software Eng., 21: 705-716.

Beizer, B., 1990. Software Testing Techniques. 2nd Edn., Van Nostrand Reinhold, New York, ISBN: 0-442-20672-0, pp: 550.

Bentley, J.E., 2005. Software testing fundamentals–concepts, roles and terminology. SUGI30, Wachovia Bank, Charlotte NC, SUGI 2005 Paper 141-30.

Bertolino, A., 2003. Software testing research and practice. Proceedings of the 10th International Workshop on Abstract State Machines, March 3-7, Taormina, Italy, 244-262.

Beydeda, S. and V. Gruhn, 2003. BINTEST-Binary search-based test case generation. Proceedings of Computer Software and Applications Conference, November 2003, Leipzig Univ., Germany, pp: 28-33.

Blackburn, M. and R. Busser, 1996. T-VEC: A tool for developing critical systems. Proceedings of the Annual Conference on Computer Assurance, (ACCA'96), IEEE Computer Society Press, pp: 237-249.

Cavarra, A., C. Crichton, J. Davies, A. Hartman, T. Jeron and L. Mounier, 2000. Using UML for automatic test generation. Oxford University Computing Laboratory, Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2000.

Chen, J., Y. Lu and X. Xie, 2008. Testing approach of component security based on dynamic fault tree. Inform. Technol. J., 7: 769-775.

Cunning, S.J. and J.W. Rozenblit, 1999. Automatic test case generation from requirements specifications for real-time embedded systems. IEEE Int. Conf. Syst. Man Cybernetics, 5: 784-789.

El-Far, I.K. and J.A. Whittaker, 2001. Model-based software testing. http://143.225.25.115/~flammini/materiale/Model-based%20Testing/ModelBased SoftwareTesting.pdf.

Firesmith, D., 2004. Prioritizing requirements. J. Object Technol., 3: 35-47.

Frohlich, P. and J. Link, 2000. Automated test case generation from dynamic models. Lecture Notes Comput. Sci., 1850: 472-491.

Gargantini, A. and C. Heitmeyer, 1999. Using model checking to generate tests from requirements specifications. ACM SIGSOFT Software Engineering Notes, 24: 146-162.

Gupta, N., A.P. Mathur and M.L. Soffa, 1998. Automated test data generation using an iterative relaxation method. ACM SIGSOFT Software Eng. Notes, 23: 231-244.

Harel, D., 1987. Statecharts: A visual formulation for complex system. Sci. Comput. Program, 8: 232-274.

Harel, D., 1988. On visual formalisms. Commun. ACM., 31: 514-530.

Heumann, J., 2001. Generating test cases from use cases. Rational Software. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf.

Huaikou, M. and L. Ling, 2000. A test class framework for generating test cases from Z specifications. Proceedings of 6th IEEE International Conference on Complex Computer Systems, Sept. 11-15, Tokyo, Japan, pp: 164-164.

Javed, A.Z., P.A. Strooper and G.N. Watson, 2007. Automated generation of test cases using model-driven architecture. Proceeding of the Second International Workshop on Automation of Software Test, May 20 - 26, Minneapolis, USA, 150-151.

Jia, X. and H. Liu, 2002. Rigorous and automatic testing of web applications. Proceedings of 6th IASTED International Conference on Software Engineering and Applications, May 2002, Honolulu, USA., pp: 654-668.

Jia, X., H. Liu and L. Qin, 2003. Formal structured specification for web application testing. Proceedings of the Midwest Software Engineering Conference, (MSEC'03), Chicago, USA., pp: 88-97.

Kancherla, M.P., 1997. Generating test templates via automated theorem proving. Technical Report, NASA Ames Research Center.

Kaner, J.D.C., 2003. What is a Good Test Case?. Florida Institute of Technology, USA.

Kim, H., S. Kang, J. Baik and I. Ko, 2007. Test cases generation from UML activity diagrams. Proceedings of 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, July 30-Aug. 1, Qingdao, China, pp: 556-561.

Korel, B., 1990. Automated software test data generation. IEEE. Trans. Software Eng., 16: 870-879.

Kosindrdecha, N. and J. Daengdej, 2009. Test case generation techniques. Technical Report 25531. Assumption University, Thailand.

Kung, D.C., C.H. Liu and P. Hsia, 2000. An object-oriented web test model for testing web applications. Proceedings of the First Asia-Pacific Conference on Quality Software, (APCQS'00), Los Alamitos, pp: 111-111.

Liu, C.H., D.C. Kung, P. Hsia and C.T. Hsu, 2001. An object based data flow testing approach for web applications. Int. J. Software Eng. Knowledge Eng., 11: 157-179.

Mead, N.R., 2008. Requirements Prioritization Introduction. SEI., Pittsburgh, Pennsylvania.

Memon, A.M., M.E. Pollack and M.L. Soffa, 1999. Using a goal-driven approach to generate test cases for GUIs. Proceedings of the 21st International Conference on Software Engineering, May 16-22, Los Angeles CA, 693-694.

Myers, G.J., 1979. The Art of Software Testing. John Wiley and Sons, NY.

NIST, 2002. The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf.

Nilawar, M. and S. Dascalu, 2003. A UML-based approach for testing web applications. M.Sc. Thesis, University of Nevada, Reno

Nilsson, R., J. Offutt and J. Mellin, 2006. Test case generation for mutation-based testing of timeliness. Electronic Notes Theor. Comput. Sci., 164: 97-114.

Offutt, A.J., Y. Xiong and S. Liu, 1999. Criteria for generating specification-based tests. Proceedings of the 5th International Conference on Engineering of Complex Computer Systems, Oct. 18-22, Washington, USA., pp: 119-119.

Pan, J., 1999. Software testing. (18-849b Dependable Embedded Systems). Electrical and Computer Engineering Department, Carnegie Mellon University.http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/presentation.pdf.

Pargas, R.P., M.J. Harrold and R.R. Peck, 1999. Test-data generation using genetic algorithms. Software Testing Verification Reliability, 9: 263-282.

Prasanna, M., S.N. Sivanandam, R. Venkatesan and R. Sundarrajan, 2005. A survey on automatic test case generation. Acad. Open Internet J., 15: 1-6.

Ramamoorthy, C., S. Ho and W. Chen, 1976. On the automated generation of program test data. IEEE. Trans. Software Eng., 2: 293-300.

Rayadurgam, S. and M.P.E. Heimdahl, 2001. Coverage based test-case generation using model checkers. http://74.125.155.132/scholar?q=cache:GvcivzIR4-wJ:scholar.google.com/+Coverage+Based+Test-Case+Generation+using+Model+Checkers.andhl=enandas_sdt=2000.

Rayadurgam, S. and M.P.E. Heimdahl, 2001. Test-sequence generation from formal requirement models. Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering, Oct. 22-24, Boca Raton, Florida, pp: 23-23.

Reza, H., K. Ogaard and A. Malge, 2008. A model based testing technique to test web applications using statecharts. Proceedings of 5th International Conference on Information Technology: New Generations, April 7-9, Las Vegas. pp: 183-188.

Ricca, F. and P. Tonella, 2001. Analysis and testing of web applications. Proceedings of the 23rd International Conference on Software Engineering, (ICSE'01), Toronto, Ontario, Canada, pp: 25-34.

Ryser, J. and M. Glinz, 2000. SCENT: A method employing scenarios to systematically derive test cases for system test. Technical Report, http://portal.acm.org/citation.cfm?id=901553.

Santiago, V., A.S.M. Do-Amaral, N.L. Vijaykumar, M.D.F., M.attiello-Francisco, E. Martins and O.C. Lopes, 2006. A practical approach for automated test case generation using statecharts. Proceedings of the 30th Annual International Computer Software and Applications Conference, Sept. 17-21, IEEE Computer Society, pp: 183-188.

Shams, M., D. Krishnamurthy and B. Far, 2006. A model-based approach for testing the performance of web applications. Proceedings of the 3rd International Workshop on Software Quality Assurance, Nov. 6, New York, USA., pp: 54-61.

Sinha, A. and C.S. Smidts, 2005. Domain specific test case generation using higher ordered typed languages from specification. Ph.D. Thesis, University of Maryland

Sommerville, I., 2000. Software Engineering. 6th Edn., Addison-Wesley, England.

Subraya, B.M. and S.V. Subrahmanya, 2000. Object driven performance testing in Web applications. Proceedings of the First Asia-Pacific Conference on Quality Software, Oct. 30-31, Hong Kong, China, 17-26.

Tierstein, L.M., 1997. Managing a designer/2000 project. NYOUG Fall'97 Conference, 1997.

Tran, H., 2001. Test generation using model checking. European Conference on Software Maintenance and Reengineering, CSMR2001, http://www.cs.toronto.edu/~chechik/courses00/csc2108/projects/4.pdf.

Tsai, W.T., X. Wei, Y. Chen, R. Paul and B. Xiao, 2005. Swiss cheese test case generation for web services testing. IEICE-Trans. Inform. Syst., 88: 2691-2698.

Turner, D.A., M. Park, J. Kim and J. Chae, 2008. An activity oriented approach for testing web applications. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, Sept. 15-19, Washington, USA., pp: 411-414.

Vijaykumar, N.L., S.V. De-Carvalho and V. Abdurahiman, 2002. On proposing statecharts to specify performance models. Int. Trans. Operat. Res., 9: 321-336.

Weyuker, E., T. Goradia and A. Singh, 1994. Automatically generating test data from a boolean specification. IEEE. Trans. Software Eng., 20: 353-363.

Wiegers, K.E., 1999. First things first: Prioritizing requirements. http://www.processimpact.com/articles/prioritizing.pdf.

William, H.C., 1988. The Complete Guide to Software Testing. 2nd Edn., QED Information Sciences, Inc., Wellesley, USA.

Wu, Y. and J. Offutt, 2002. Modeling and testing web-based applications. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.4485andrep=rep1andtype=pdf.

Wu, Y., J. Offutt and X. Du, 2004. Modeling and testing of dynamic aspects of web applications. Technical Report ISE-TR-04-01, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.9723andrep=repl andtype=pdf

Yang, J.T., J.L. Huang, F.J. Wang and W.C. Chu, 1999. Constructing control-flow-based testing tools for web application. 11th Software Engineering and Knowledge Engineering Conference (SEKC'99), June 1999.

Yang, J.T., J.L. Huang, F.J. Wang and W.C. Chu, 2002. Constructing an object-oriented architecture for web application testing. J. Inform. Sci. Eng., 18: 59-84.