Asian Journal of Information Technology 15 (21): 4276-4290, 2016
ISSN: 1682-3915
© Medwell Journals, 2016

An Optimized Approach for Automated Test Case Generation and Validation for Uml Diagrams

Roaa Elghondakly, Sherin Moussa and Nagwa Badr
Department of Information Systems, Faculty of Computer and Information Sciences,
Ain Shams University, Cairo, Egypt

Abstract: Software testing is accounted to be an important phases in software development life cycle in terms of cost and manpower. Consequently, many studies have been conducted to minimize the associated cost and human effort to fix bugs and errors and to improve the testing process's quality by generating test cases at early stages. However, most of them considered only one type of behavioural diagrams with a lot of human intervention. In this study, an optimized automated approach for generic test case generation was proposed. It is considered as generic in terms of it can be applied on different types of behavioural diagrams (i.e. activity diagram, state diagram, uses case diagram, etc.) for multi-disciplinary domains. While the automation process is used to generate test case with minimum human intervention which will consequently help to minimize total cost. Testing process is considered the key to success of any software. An optimized test case generation approach therefore will be very useful. As a result an optimization technique has been applied to optimize the generated test cases to ensure the quality of results. Accordingly, the proposed approach merges model-based testing with search-based testing to automatically generate test cases from different behavioural diagrams, i.e. use case, activity, etc. Moreover, the proposed approach uses text mining and symbolic execution methodology for test data generation and validation where a knowledge base is developed for multi-disciplinary domains.

Key words: Test case generation, model-based testing, search-based testing, test case optimization, test case validation, UML diagrams

INTRODUCTION

Software testing is an essential part in software development life cycle. The development life cycle total cost is considerable to be high. For this reason, it is important to minimize this cost and the human effort to fix bugs and errors, as well as to improve the quality of the testing process by automating it. Test automation is the process of using separate software to manage and evaluate the fulfillment of test cases and compare the expected outcomes with the generated ones. It can be performed with less human intervention saving total cost. Moreover, it can add further testing which would be complicated to perform manually. Testing may be performed in the last phase of the software development life cycle or at an early stage. If the testing process is carried out at early phase of the development life cycle, (i.e., requirements and design phases), as in the case of model-based testing, it will guarantee higher test quality, early specification and review of system behavior (Hartmann *et al*., 2004; Prasanna *et al*., 2011; Prasanna and Chandran, 2011; Schieferdecker, 2012). However, if it is carried out in the last phase as in the case of code-based testing, it will not be effective as in early stages (Ye *et al*., 2009). In this case, this will generate enormous errors as soon as the code is completed where it demanded a lot of code correction and modification. For this reason, model-based testing was preferable rather than code-based testing to generate test cases from UML diagrams (behavioral diagrams) during the design phase.

The Unified Modeling Language (UML) is divided into two main parts; structural diagrams, (i.e., class diagram) and behavioral diagrams, (i.e., activity, use-case and state diagrams). Structural diagrams are used to show the structure, style or design of the software while behavioral ones are used to clarify the steps in which the software will pass through until it reaches the desired output. In other words, it shows the flow of events. However, the optimization process has different techniques as reducing the number of test cases, test cases prioritization, as well as minimizing time, increasing performance, maximizing the quality of outcomes, etc. A search-based testing technique is used in this approach.

Corresponding Author: Roaa Elghondakly, Department of Information Systems, Faculty of Computer and Information Sciences, Ain Shams University, Cairo, Egypt

Search Based Software Testing (SBST) (McMinn, 2011; Iqbal *et al*., 2012) is a branch of Search Based Software Engineering (SBSE), in which optimization algorithms are used to allow dynamic search for test data to maximize the achievement of test goals and minimize testing costs. As a reduced set of test cases is generated and executed, the total testing time will be minimized as well as the total cost. However, the testing process counters some problems as high cost and time.

In this study, a new generic, optimized and automated model with minimum human intervention is proposed for automatic test case generation where model-based testing and Search-based testing are combined to generate the optimum test case. In order to ensure the correctness of these generated test cases, a test data generation technique (Korel, 1990; Veanes *et al*., 2008; Harman *et al*., 2011; Tahbildar and Kalita, 2011; Shahbaz *et al*., 2015) will be applied to validate the test cases and to ensure that the previously generated test case when being executed will meet the expected output. The study is organized as follows.

Literature review: As model-based software development becomes more important, the relevance of model-based test cases generation increases as well. The ability to cover multiple UML models is appealing. In the last decade, test case generation became an important research topic. Many researchers use behavioural diagrams (Use Case, Activity and State) in order to generate test cases. However, most of the previous studies used different techniques but they encounter some limitations, as not being generic (Kim *et al*., 2007; Chen *et al*., 2008; Chimisliu and Wotawa 2012, 2013) in which they are limited to only one type of diagrams in each single method. In addition, some are presenting manual approaches (Heumann, 2001; Gutierrez *et al*., 2006; Kundu and Samanta, 2009; Nayak and Samanta, 2011; Pechtanum and Kansomkeat, 2012; Sumalatha and Raju, 2012) which are not time saving. It is possible to build automatic tools following their approaches which are expected to reduce cost of software development and improve software quality. Whereas the alternating variable method used in (Samuel *et al*., 2008) in generating test data does not provide globally optimal solution as well as Swain *et al*. (2012) since it does not provide optimization. Moreover, developing complicated methods to generate test cases is considered a limitation as in Gnesi *et al*. (2004) and Santiago *et al*. (2008).

The studies in Swain and Mohapatra (2010), Boghdady *et al*. (2011a, b) and Pechtanun and Kansomkeat (2012) used the activity diagram to generate test cases. A dependency table for the diagram was generated as well as a corresponding graph. The graph was then parsed to find all possible paths between the start and end nodes. Researchers of Kundu and Samanta (2009) and Chen and Li (2010) stored the activity diagram as XML Metadata Interchange (XMI) files and then retrieved the required information from the XMI file. Mapping rules were then applied to generate a corresponding graph from which test paths and test cases were generated respectively. Li *et al*. (2013) built an algorithm to generate test sequences by applying Euler circuit algorithm to decrease the number of generated test cases. Ray *et al*. (2009) on the other hand reduced the number of generated test cases by applying conditioned slicing on a predicate node of the graph generated from the activity diagram. Sumalatha and Raju (2012) integrated the activity diagram with the sequence diagram. They used the corresponding graphs generated from both diagrams to generate test cases. As for the state diagrams, the one in Kansomkeat and Rivepiboon (2013) was transformed to flow graph to help in the test cases generation process. Genetic algorithms were used in the test case generation process as in Doungsa *et al*. (2007) and Shirole *et al*. (2011). Concerning use-case diagrams, the use-case diagram was transformed to activity diagram in Gutierrez *et al*., 2007a, b) where test cases were generated by the different methods as stated earlier.

In search-based software engineering, several researches (McMinn, 2004) have studied some algorithms as Hill Climming, Simulated Annealing and Genetic algorithms. The first two algorithms were used with the early researches but they were used with reference to only one elected solution at one time. For this reason, using Genetic Algorithms (GAs) (Doungsa *et al*., 2007) became more efficient as they are global searches and can sample more than one point at the same time in the search space. It is easy to understand and solves problems with multiple solutions. Although, GAs have many advantages, they counter some disadvantages, i.e., specific optimization problems cannot be solved. This happens because fitness functions are not known. Besides, there is no assurance that they will find a global optimum. In addition, genetic algorithm real time applications are limited due to random solutions and convergence. Researchers have proved that Ant Colony Optimization (ACO) (Dorigo *et al*., 2006; Li and Li *et al*., 2007) is better than GAs, as it overcomes some of the disadvantages found by GAs. GAs use global search technique which can sample more than one point at the same time in the search space. ACO is considered better than other global optimization techniques for TSP as genetic algorithms, neural network techniques and simulated annealing. In addition, it reserves an entire

colony memory instead of the previous generation only. Moreover, the combination of colony memory and the random path selection makes it less affected by poor initial solutions. ACO can be used in dynamic applications where it can be adjusted to real time application's changes as it runs continuously.

Looking into a software requirement, either functional or non-functional, it identifies a system quality, capability, characteristics and necessary attributes, as well as it describes a utility to an internal user, a customer organization or other stakeholder. Requirements are mostly written using use cases model, (i.e., a textual description for every use case flow of events). This textual description can be adopted easily. However, it is too complicated to perform further processes on them, such as generating test cases. Many researchers use either manual or automatically textual requirements to build state-charts diagrams (Frohlich and Link, 2000). Some approaches translate natural language requirements automatically to a logical or systematic format to generate test cases automatically from those textual requirements. Others neglect textual requirements and replace them with a use case diagram which is later transformed to an activity diagram. Whereas some approaches, convert natural language requirements to a flow graph (diagram), then extract test paths from it by applying path coverage methods. Other approaches that perform coverage criteria combine Boolean propositions on natural language requirements. To summarize, most of the researches on this field performs natural language processing techniques on textual requirements to facilitate the process of test cases generation.

Finally, despite of the importance of all UML diagrams and the enrichment they provide to better describe systems, most of the researches generate test cases only either from activity diagrams, sequence diagrams or both (Li *et al.*, 2007; Shirole and Kumar, 2010; Boghdady *et al.*, 2011a, b; Shanthi and Kumar, 2012). The approaches in Swain and Mohapatra (2010), Sumalatha and Raju (2012) combine activity and sequence diagrams together to generate test cases.

Although, many approaches have been proposed to automate test cases generation, it was found that most of related work studies use only one type of UML diagrams in their approaches. Only few ones combine two diagrams. Some are manual and others are automated. However, the main contribution of our study is that we present a novel comprehensive approach that can automatically generate test cases from different UML diagrams (activity, state and use-case, etc.) for multi-disciplinary domains. In addition, it combines model-based testing with search-based testing techniques in order to generate the optimum test case.

MATERIALS AND METHODS

The proposed system architecture: In this study, an optimized, generic and automated model with minimum human intervention is proposed. The model is considered to be generic since it is capable of generating test cases from any behavioural diagram regardless its type (use-case, activity, state, etc.) for multi-disciplinary domains, i.e., ATM systems, registration systems, etc. Search-based testing techniques are applied to find the optimum test cases from the generated ones.An enhanced Ant Colony Optimization (EACO) technique is proposed as a search-based testing technique. The Ant Colony Optimization (ACO) algorithm is an optimization algorithm that is used to find the optimum path in a graph. As in (Dorigo *et al.*, 2006; Kundu and Samanta, 2009) it takes an inspiration from the foraging behaviour of some ant species. These ants deposit pheromone on the ground in order to mark some favourable path that should be followed by other members of the colony.This research proposes an Enhanced Ant Colony Optimization algorithm (EACO) that is used to find the optimum test case to reduce the execution time.

Although, behavioural diagrams do not have weighted edges, this algorithm requires weighted edges. In which, edges' weights are used to calculate the probability of which node should be selected next by each ant. For example, now ant 1 stands at the first node which is connected with >1 node. In order to help this ant to know which node to go next, therefore the distance between nodes are used as edges' weights to enable the ant select its next node in the path. In most cases, UML diagrams generate graphs having un-weighted edges. Generated weighted graph used to show that node X is connected to node Y only. Accordingly, the generated graph is enhanced by adding weights to edges. Weights are given to edges according to the maximum number of nodes connected to node X and node Y. The maximum number of nodes is used to ensure that the edge's weight will be >1. Therefore, each node has at least one child to guarantee the continuous connection between nodes (continuous paths from source to destination). A reduced set of test cases is then generated having the test paths with the least weights. This set is then used to find the optimum test case according to the number of decision nodes it passes through. As the number of decision nodes in a path decreases, the probability to be the optimum test path increases.

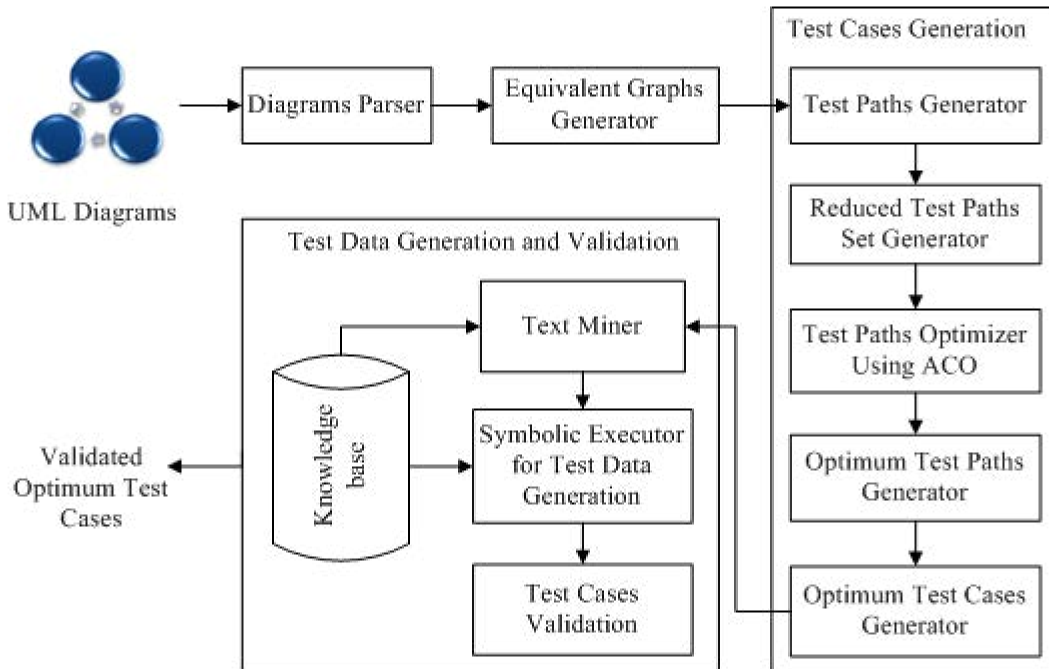

Fig. 1: The Proposed system architecture

The proposed model automatically generates test cases from behavioural diagrams by combining model-based with search-based testing. As the modelling process itself requires a lot of human intervention, the automatic generation of test cases takes place after having the behavioural diagram as an input. The proposed model consists of four main modules as shown in Fig. 1: diagrams parser, equivalent graphs generator, test case generation and test data generation and validation. The proposed model is a generic model that takes different behavioural diagrams as an input. The following sub-sections present a brief description for the functionality of each module in the proposed approach.

Module 1; Diagrams parser: This module is responsible for parsing any type of behavioural diagrams to generate a corresponding excel file. The resultant file contains all the diagram's data, including state, activity or use-case names, IDs, relations between them, etc. which is then stored into the database as records. A sample of three different behavioural diagrams and their equivalent Excel sheets are shown in Fig. 2-7.

Module 2; Equivalent graphs generator: Once the equivalent Excel sheet is generated, a graph is constructed to represent the associated behavioural diagram. The constructed graph represents the states (or activities) as nodes and the relations between them as edges/arcs. Consequently, this graph is used to find all paths between any two nodes using Depth First Search (DFS) as DFS uses lower memory in comparison with that used by Breadth First Search.

Module 3; test cases generation: Module represents the core of the proposed model. It includes several sub-modules; test paths generator, reduced test paths sets generator, reduced test paths optimizer using Ant Colony Optimization (ACO) (Dorigo *et al*., 2006), as it overcomes some of the disadvantages found by GAs as mentioned before in section II, optimum test paths generator and finally, optimum test cases generator. The previously constructed graph is used to generate test paths. Many algorithms have been studied to parse graphs to find all paths between start and end nodes, such as Depth First Search and Breadth First Search. In the proposed approach, Depth First Search is applied since it consumes less memory in comparison with that used by Breadth First Search where it is not necessary for each level to store all child pointers (Korf, 1985). The generated paths are sequences of nodes IDs in which each sequence represents a single path from the start node to the end node.

A reduced set of test paths is then generated and optimized to generate the optimum test paths. The generated test paths are filtered by selecting the test paths with the smallest path weight regardless the number

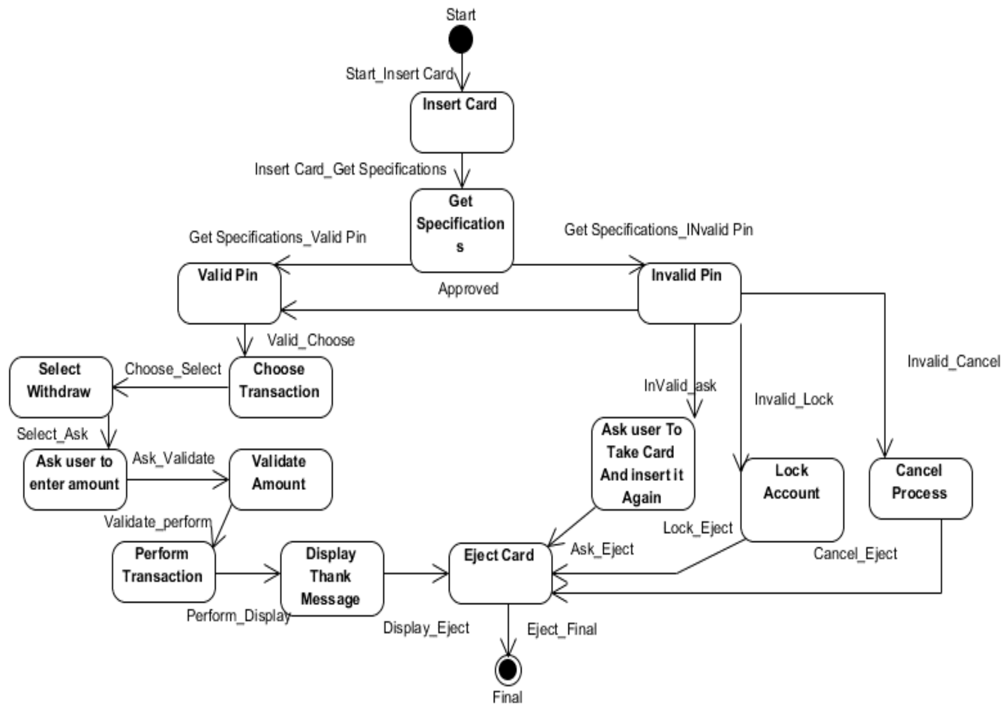

Fig 2: State diagram example

	A	B	C	D
1	**ID**	**Name**	**Source**	**Target**
2	1	StateDiagram		
3	4	Display Thank Message		
4	3	Perform Transaction		
5	8	Eject Card		
6	12	Select Withdraw		
7	13	Ask user to enter amount		
8	17	Final		
9	20	Start		
10	21	Insert Card		
11	26	Get Specifications		
12	28	Choose Transaction		
13	30	Ask user To Take Card And insert it Again		
14	33	Valid Pin		
15	35	Lock Account		
16	37	Cancel Process		
17	40	Invalid Pin		
18	67	Validate Amount		
19	2	Perform_Display	3	4
20	7	Display_Eject	4	8
21	11	Select_Ask	12	13
22	19	Start_Insert Card	20	21
23	25	Insert Card_Get Specifications	21	26
24	32	Get Specifications_Valid Pin	26	33
25	39	Get Specifications_INvalid Pin	26	40
26	44	Approved	40	33
27	48	InValid_ask	40	30
28	51	Invalid_Lock	40	35
29	53	Invalid_Cancel	40	37
30	55	Valid_Choose	33	28
31	57	Choose_Select	28	12
32	59	Ask_Eject	30	8
33	61	Lock_Eject	35	8
34	63	Cancel_Eject	37	8
35	65	Eject_Final	8	17
36	69	Ask_Validate	13	67
37	71	Validate_perform	67	3

Fig.3: Equivalent excel sheet for the state diagram example

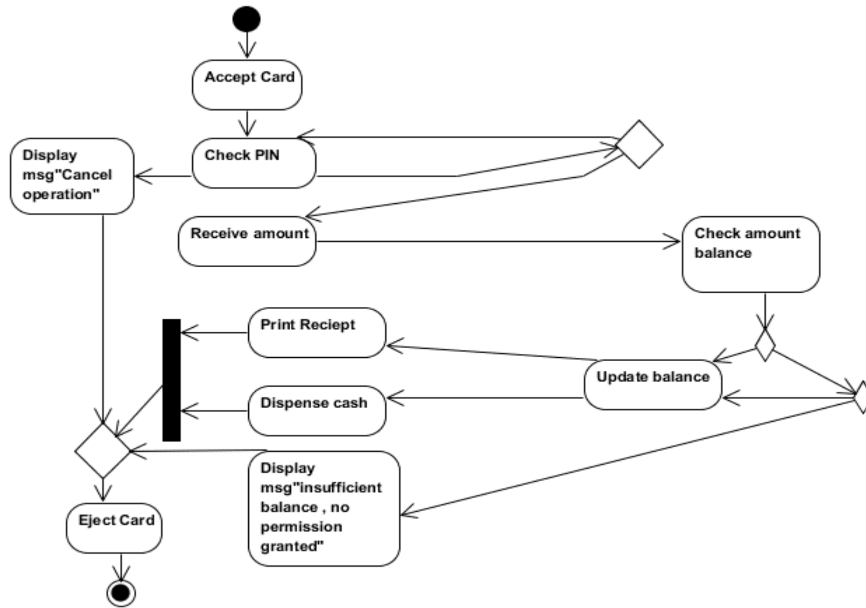

Fig. 4: Activity diagram examp l

ID	Name	Source	Target
1	Type : ActivityDiagram		
2	Start		
4	Accept Card		
6	Check PIN		
8	Receive amount		
10	Display msg"Cancel operation"		
12	Print Reciept		
14	Dispense cash		
16	Display msg"insufficient balance , no permission granted"		
18	Eject Card		
25	Final		
37	Fork Node		
29	Decision Node 4		
47	Decision Node 1		
55	Check amount balance		
59	Decision Node 2		
61	Update balance		
71	Decision Node 3		
20		2	4
23		4	6
27		18	25
31		29	18
33		6	10
35		10	29
39		12	37
41		14	37
43		37	29
45		16	29
49		6	47
51		47	6
53		47	8
57		8	55
63		59	61
65		55	59
67		61	12
69		61	14
73		59	71
75		71	61
77		71	16

Fig. 5: Equivalent excel sheet for the activity diagram example

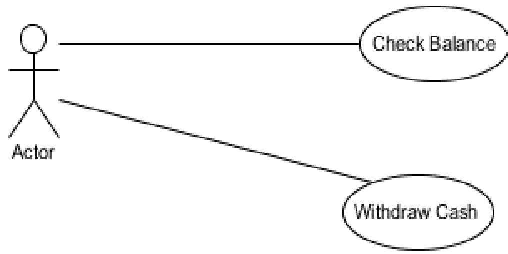

Fig. 6: Use Case Diagram Example

A	B	C	D
ID	Name	From	To
1	UseCaseDiagram		
2	Withdraw Cash	UseCase	
4	Check Balance	UseCase	
7	Customer	Actor	
6		7	4
13		7	2

Fig. 7: Equivalent excel sheet for the use case diagram example

of nodes. Accordingly, test cases are generated automatically from the generated test paths where the sequences of nodes IDs become sequences of meaningful test cases as each node in the sequence represents the name of a use case, state or activity from the original diagram. Test cases presentation makes the test paths more useful for the user (tester), as it converts idiot IDs to meaningful states or processes names. Finally, the optimum test case is selected from the created test cases. As for test cases optimization, An Enhanced Ant colony optimization EACO technique is applied based on the advantages discussed earlier in section II. The optimum test case is selected according to the path with the highest pheromone value (Dorigo *et al*., 2006) and the shortest distance where the pheromone value and the distance are inversely proportional.

Module 4; test data generation and validation: This module is concerned with validating the optimum test case. One of the test data generation techniques is the Symbolic Evaluation (also referred to as Symbolic Execution). It involves executing a program using symbolic values of variables instead of actual values (Korel, 1990; Veanes *et al*., 2007; Harman *et al*., 2011; Tahbildar and Kalita, 2011; Shahbaz *et al*., 2015). Symbolic evaluation technique is used to simulate symbols to act as an input to the system under test to find the expected output according to those inputs where it can be performed either in a forward traversal (forward substitution) or a backward traversal (backward substitution) for the path. In the proposed approach, forward substitution is applied, storing what a programmer does while tracing an execution path to aggregate the symbolic values of variables. Thus, it has to be carried out on every statement, while saving all the intermediate symbolic values of variables. During the test data generation process, values are assigned to the input variables as a first step. Once the values of variables are known, a symbolic value is then given by the most recent instance of the array for each array element that has just been assigned a value. The success of this technique depends on its ability to assign values to simple variables and array variables separately.

Most of the previous works studied test data generation techniques as a code -based source code rather than model-based. This is due to the easiness encountered in the case of code, as it has equations and conditions that help the process of test data generation (Clarke, 1976; Xing *et al*., 2015). The previously generated optimum test case is passed to the proposed Text Miner sub-module to parse it and generate its corresponding grammatical tree. The Text Miner module uses the Part-of-Speech (POS) tagging and parsing approach for the natural language processing required at this stage. POS tagging assigns a POS tag to each word in the sentences in order to indicate whether it is a noun, verb, adverb, adjective, etc. (Kulkarni and Joglekar, 2014). The part-of-speech POS category names can be identified by applying a Part-of-Speech (PoS) POS tagger and thereby then ignoring any non-verb tokens (Shahbaz *et al*., 2015). All detected verbs are saved into a knowledge base. Forward substitution symbolic execution is then applied on each stored verb to be validated. Consequently, validated optimum test cases are generated. The proposed model is not restricted to a specific domain for validation.

Finally, the proposed architecture shows that model-based testing, presented by generating test cases from UML diagrams, is combined with search-based testing, in which optimization algorithm (ACO) is used to generate optimum test case. In addition, text mining using POS tagging approach is used to detect the verbs from the optimum test case. Moreover, the forward substitution method is applied in our proposed architecture for symbolic execution to simulate inputs (validation sentences) to verbs in order to validate the optimum test case.

RESULTS AND DISCUSSION

In order to test and evaluate the proposed approach, it is applied on the commonly used ATM scenario (Chen and Li, 2010). Several behavioural diagrams such as

activity, state, use-case diagrams are used representing the system's behaviour to describe the sequence of events for the given scenario. The proposed approach takes place after having the behavioural diagrams been modelled, which ensures saving time as the modelling process itself consumes a lot of time.The processing of our proposed approach is presented as follows:

- Input different behavioural diagrams (state, activity, use-case)
- Generate the corresponding Excel file for each diagram
- Construct the corresponding graph for each diagram
- Parse the previously generated graph using Depth First Search algorithm (DFS) to generate all possible paths
- Generate a set of reduced paths
- Apply Enhanced Ant Colony Algorithm (EACO)
- Optimize the generated test case
- Perform symbolic execution for test data generation
- Validate the optimum test case

Figure 2-7 represent three behavioural diagrams and their equivalent Excel sheets used in the performed experiments. The state diagram shown in Fig. 2a and b are for the PIN verification. The automation process takes place after having the behavioural diagrams been modelled which will ensure saving time as the modelling process itself consume a lot of time. The activity diagram shown in Fig. 3a and b are for the withdraw process where the use case diagram shown in Fig. 4a and b are for the ATM scenario. In case of the use-case diagram is converted to several activity diagrams derived from the flow of events for each use case according to the number of use cases. Each activity diagram then passes through the proposed system where the outputs of all the activity diagrams are combined together as a result of the use case diagram.

States (activities) are represented by nodes, while transitions between nodes are represented by directed arcs where states, activities and use cases names must be unique within the scope of a diagram. The proposed system parses the UML behavioural diagrams to generate their corresponding Excel sheets as explained in this study. The generated Excel file contains 4 attributes: ID (unique ID for each node or relation), Name (contains the name of a node or a relation) where this attribute may have null values in case of relations (edges) as not all relations have names. Source and Target attributes represent the relation (edge) between any two nodes. Source and target have the ID as a value to show that there is a relation between certain nodes according to their IDs. These two attributes may have null values as well if there is no relation as in case of nodes names. For activity diagrams, fork nodes are handled in which all nodes connected to a fork node must be combined and represented together before exiting such fork node and moving to the next node. A graph is constructed using the data retrieved from the Excel file and then parsed to find all paths using Depth First Search (DFS) algorithm. An enhanced Ant Colony Optimization (EACO) algorithm is proposed to find the optimum test case. EACO is proposed to select the next node modified using the enhanced graph.

The optimum test case is then parsed where its grammatical tree is generated to find all verbs in this test case. Text mining using the POS tagging approach is applied on the verbs of the optimum test case for test data generation. Forward substitution symbolic execution technique is used to validate the optimum test case. After developing the proposed architecture, the generated output for the previously presented UML diagrams is shown in Fig. 8-10.

The first section "All Paths" represents all possible test paths between the start and end nodes (states, activities). The second section "All test cases" shows the test cases with its names. The "Weights" section displays the total weight for each path. "Set of Reduced Paths" section displays the list of reduced test paths. "Set of Test Cases for Reduced Paths" section presents the textual representation for each test path from the Set of Reduced Paths. The "Optimum Reduced Test Path" section shows the optimum test path as illustrated previously, while "Optimum Reduced Test Case" section shows the textual representation of the test case having the optimum test path. The last two sections represent the "Optimum Test Case Symbolic Execution data" which displays the list of detected verbs from the optimum test case with their associated test data after applying forward substitution symbolic execution. Whereas "Optimum Test Case Validation" section displays the list of test cases to validate the optimum test case.

Different metrics can be applied to evaluate the quantity and quality of the test case generation process from UML diagrams, including the complexity of the generation algorithm, time, coverage criteria and the number of generated test cases. We have used time and cyclomatic complexity metrics in order to evaluate our proposed approach. In the following sub-sections, we present the results encountered while assessing our proposed approach in terms of these metrics.

Cyclomatic complexity: Some software testing concepts are used, like statement coverage, branch coverage and

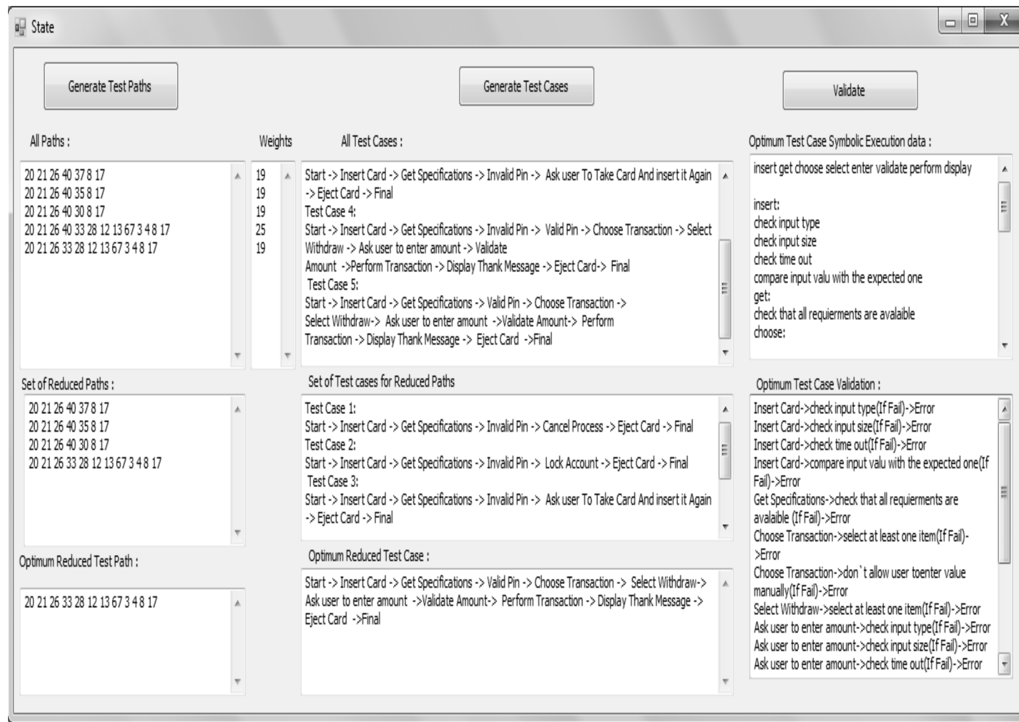

Fig. 8: GUI of the proposed system in case of a state diagram

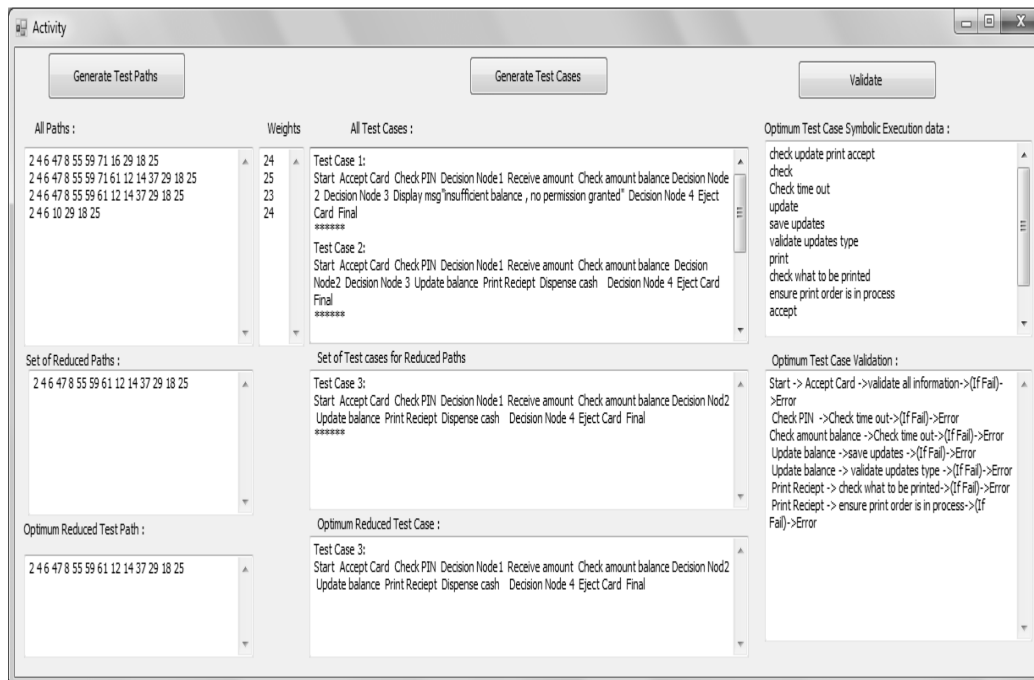

Fig. 9: GUI of the proposed system in case of an activity diagram

path coverage (McQuillan and Power, 2005; McCabe, 1976). In statement coverage, the test case is executed in a way that every statement (node) of the diagram is executed at least once. In branch/decision coverage,

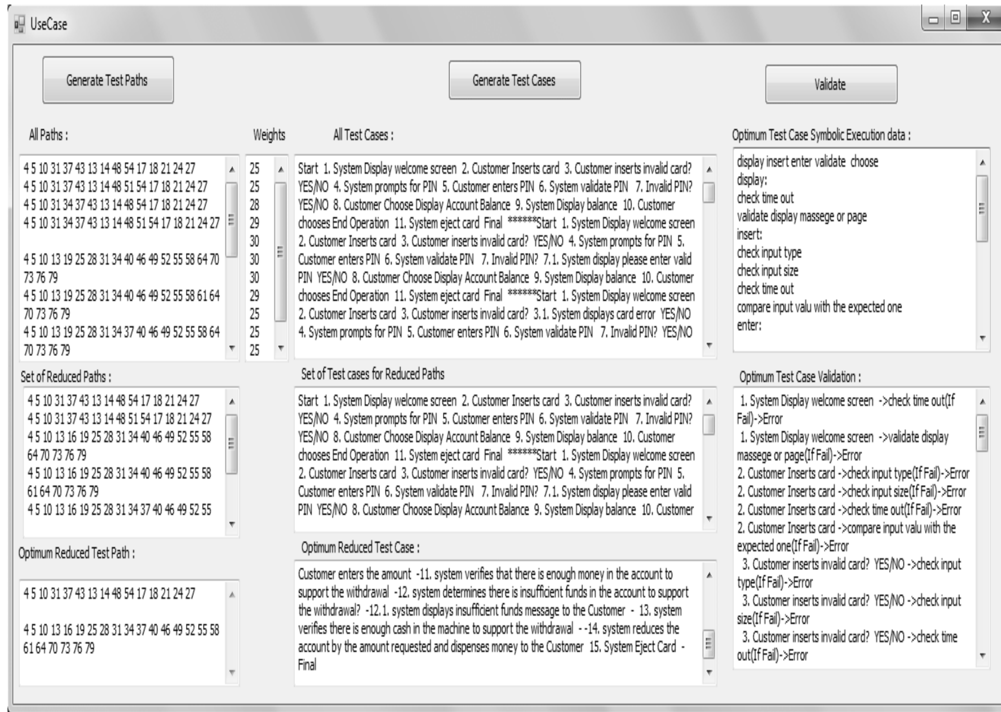

Fig.10: GUI of the proposed system in case of a use case diagram

the test coverage criteria requires enough test cases, such that each condition in a decision takes on all possible outcomes at least once and each point of entry to a program or subroutine is invoked at least once. Whereas Path Coverage executes each test case in such a way that every path is executed at least once. The test cases are prepared based on the logical complexity measure of a procedural design. In path coverage of testing, every statement in the program is guaranteed to be executed at least once.

Cyclomatic Complexity is used to arrive at a basis path (path coverage) (McQuillan and Power, 2005; McCabe, 1976). It is a measure used to indicate how much a program is complex quantitatively. Cyclomatic complexity is a size indicator that measures the number of logical paths in a module. It also indicates the minimum number of tests needed to forecast high reliability. It helps in computing the minimum number of test paths that has to be covered from the UML diagrams, especially activity diagrams (Boghdady *et al*., 2011a, b). Cyclomatic complexity is applicable for diagrams with different sizes. It is independent on the number of nodes. In addition, it is applied mainly on the diagrams with graph data structure as in the case of the activity diagrams. Cyclomatic complexity can be measured as shown in Eq. 1 and 2:

$$V(G) = E-N+2 \quad (1)$$

Where:

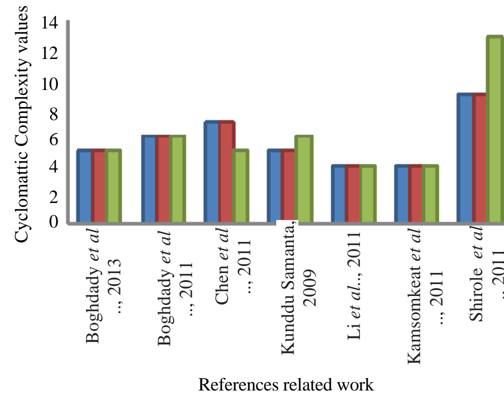

Fig. 11: Cyclomatic Complexity Comparison

E = Graph edges number
N = Graph nodes number

$$V(G) = P + 1 \quad (2)$$

where, p is the number of predicate (decision) nodes contained in the flow graph G.

$$\text{Branch coverage} \leq \text{Cyclomatic complexity} \leq \text{Number of paths} \quad (3)$$

The cyclomatic complexity was computed for different examples from related work studies. Fig. 11.

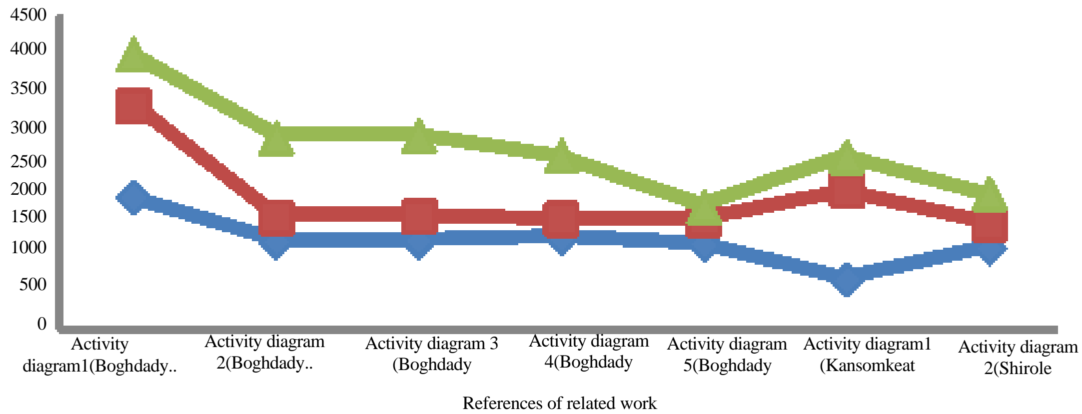

Fig. 12: Time comparison

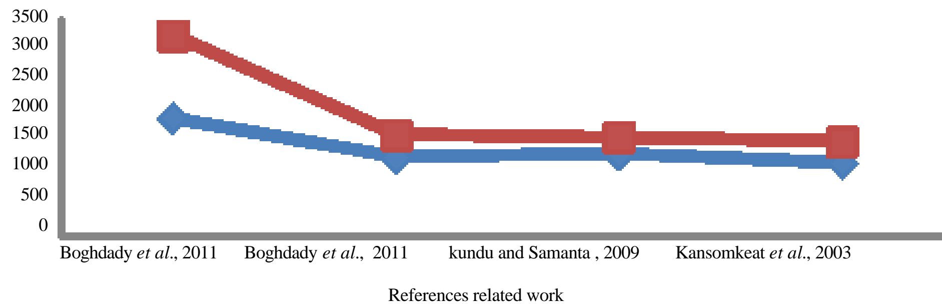

Fig. 13: Time comparison with other approaches

shows the number of generated test paths against the cyclomatic complexity technique. The blue bar represents the proposed system output, the red bar represents expected output while the green bar shows the related work output. The X-axis represents different examples from the different related work studies with variant sizes are used, while y-axis represents the Cyclomatic complexity values. In which comparison between cyclomatic complexity measurements between the proposed system the related work meets coverage criteria (statement, branch and path). For example: In (Boghdady *et al*., 2011) the expected output is V(G) = 16-13+2=5, The proposed system output is V(G) =16-13=5, The related work output is V(G) =16-13 = 5. In Boghdady *et al*. (2011) the expected output is V(G) = 6, The proposed system output is V(G) = 6. The related work output is V(G) = 6. In Kundu and Samanta (2009) the expected output is V(G) = 7, the proposed system output is V(G) = 7 The related work output is V(G) = 5.

The previous examples show that the proposed system output is equal to the expected values, which guarantees that the proposed approach achieves the minimum number of tests needed to forecast high reliability.

Time: The time encountered to automatically generate the test cases is used to evaluate our proposed approach. Figure 12 illustrates the time elapsed in milliseconds when applying the proposed approach to some examples from related work studies, in comparison with the time taken by these studies when applying Depth First Search (DFS) and Breadth First Search (BFS). Y-axis represents the time in milliseconds while x-axis represents different types of behavioural diagrams.

The proposed model is applied to different behavioural diagrams as in Kansomkeat and Rivepiboon (2013), Kundu and Samanta (2009), Chen *et al*. (2010), Boghdady *et al*. (2011a, b), Shirole *et al*. (2011) and Li *et al*. (2013). It is found that BFS consumes a lot of time as it uses huge memory to store all of the child pointers for each level while DFS uses less memory. Consequently, time is reduced. As for the proposed approach, an optimization algorithm is enhanced to reduce time effectively. Accordingly, optimized test cases are generated which helps in minimizing the total execution time as well as the total cost. Figure 13 illustrates the time used in the milliseconds

Table 1: Comparison between the proposed system and the related work

References	UML type	Automated	Generate test paths	Generate test cases	Generate reduced set of test paths	Generate reduced set of test paths	Generate optimum test path	Generate optimum test case	Apply test data generation technique	Generate optimum test case validation
Proposed system	Activity, state Use-case	✓	✓	✓	✓	✓	✓	✓	✓	✓
Swain and Mohapatra (2010)	Activity with Sequence	✓	-	✓	-	-	-	-	-	-
Hettab *et al*.(2013)	Activity	✓	✓	-	-	-	-	-	-	-
Boghdady *et al*. (2011)	Activity	✓	✓	✓	-	-	-	-	-	✓
Boghdady *et al*. (2011)	Activity	✓	✓	✓	✓	✓	-	-	-	✓
Nayak and Samanta (2009)	Activity	✓	✓	✓	-	-	-	-	-	-
Chen *et al*. (2010)	Activity	✓	✓	✓	-	-	-	-	-	-
Kundu and Samanta (2009)	Activity	✓	✓	✓	-	-	-	-	-	-
Chen *et al*. (2008)	Activity	✓	✓	✓	-	-	-	-	-	-
Li *et al*. (2013)	Activity	✓	✓	✓	✓	✓	-	-	-	-
Pechtanum *et al*. (2012)	Activity	-	✓	-	-	-	-	-	-	-
Ray *et al*. (2009)	Activity	✓	✓	✓	-	-	-	-	-	-
Sumalatha and Raju (2013)	Activity	✓	✓	✓	✓	✓	-	-	-	-
Kim *et al*. (2007)	Activity	-	✓	-	-	-	-	-	-	-
Sumalatha and Raju (2012)	Activity	✓	✓	✓	-	-	-	-	-	-
Kansomkeat and Rivepiboon (2013)	State	✓	✓	✓	-	-	-	-	-	-
Samuel *et al*. (2008)	State	✓	✓	✓	✓	✓	-	-	✓	-
Shirole *et al*. 2011)	State	✓	✓	✓	-	-	-	-	-	-
Santiago *et al*. (2008)	State chart, finite state behavioural	✓	✓	✓	-	-	-	-	-	-
Gnesi *et al*. (2004)	State	✓	✓	✓	-	-	-	-	-	-
Swain *et al*. (2012)	State	✓	✓	✓	-	-	-	-	-	-
Chimisliu and Wutawa (2013)	State	✓	✓	✓	✓	✓	-	-	-	-
Chimisliu and Wutawa (2012)	State	✓	✓	✓	-	-	-	-	-	-
Li *et al*. (2007)	State	✓	-	-	-	-	✓	-	-	-
Doungsa *et al*. (2007)	State	✓	✓	✓	-	-	-	-	✓	-
Chen *et al*. (2010)	Use-Case	✓	✓	✓	-	-	-	-	-	-
Nebu	Use-Case	✓	✓	✓	-	-	-	-	-	-
Gutiérrez *et al*. (2006)	Use-Case	✓	✓	✓	-	-	-	-	-	-
Gutierrez *et al*. 2007)	Use-Case	✓	✓	✓	-	-	-	-	-	-
Heumann (2001)	Use-Case	-	✓	✓	-	-	-	-	✓	-
Sarma and Mall 2007)	Use-Case,	✓	-	✓	-	-	-	-	-	-
Gutiérrez *et al*. (2007)	Use-Case	✓	✓	✓	-	-	-	-	-	-

when comparing the test execution time of the proposed approach with other approaches as DFS and BFS. As for the other approaches, it is notable that BFS consumes a lot of time, since it uses huge memory to store all of the child pointers for each level, while DFS uses less memory. Consequently, time is reduced. As for the proposed approach, an optimization algorithm is enhanced to reduce time effectively. Accordingly, optimized test cases are generated which helps in minimizing the total execution time, as well as the total cost. In Fig. 13 the Y-axis represents a time comparison in milliseconds while x-axis represents different types of behavioural diagrams. The blue line represents our proposed approach whereas the red one is for the related work studies. The proposed model is applied to different behavioural diagrams as in Kundu and Samanta (2009), Boghdady *et al*. (2011a, b). Boghdady *et al*. (2011a, b) and Shirole *et al*. (2011). Table 1 demonstrates the comparison between the proposed model and the related work according to different criteria.

The previous result proves that the proposed approach meets hybrid coverage criteria (statement, branch and path). It validates the number of generated test cases by meeting cyclomatic complexity coverage. It generates the optimum test path which when being executed, it saves time cost. In addition, when being compared to other approaches in measuring the total time elapsed to generate test cases, it proves time saving, especially after using the optimization technique. Moreover, Table 1 states that the proposed approach is generic, automated, optimized, comprehensive approach in comparison with the related work studied in the field of research.

CONCLUSION

Software testing is an essential phase in software life cycle where it consumes a lot of money and effort in fixing bugs. As the automation of test case generation process helps in minimizing human intervention. Therefore, it is

mandatory to automate this process in order to minimize time and cost. In this study, a generic, optimized and automated model with minimum human intervention is proposed. Generic as it generates test cases from any UML diagram regardless its type (use-case, activity, state, etc.) for multi-disciplinary domains. Then it parses them to generate the test paths then the test cases automatically. For each diagram, an enhanced weighted graph is constructed where an EACO (Enhanced Ant Colony Optimization) technique is proposed that selects the next node modified using the enhanced graph. This enhancement is because of the graphs generated from UML behavioural diagrams are un-weight. The proposed model requires weighted graph, accordingly, weights are added to edges. These weights are calculated to be the maximum number of children between any two nodes forming an edge. The maximum number is used to ensure that each node has at least one child, which guarantees the continuous connection between nodes. Therefore, DFS is used to parse the enhanced graph and find all test paths between initial and final nodes. DFS is used as it uses less memory in comparison to that used by BFS, as it is not necessary for each level to store all the child pointers. EACO (Enhanced Ant Colony Optimization) algorithm is applied on the generated set of test cases to find the optimum one (the test case with the shortest path and highest pheromone value). Test data generation technique is used to validate the optimum test case.

Finally, the proposed model is a combination between model-based and search-based testing techniques. It can be used to generate test cases automatically from different behavioural diagrams, as it can be used for state, activity, use-case diagrams. In addition, it meets coverage criteria (statement coverage, branches coverage and paths coverage). Not only optimized test cases are generated but also a test data validation technique is applied to ensure the correctness of the optimum generated test cases. In conclusion, the proposed model proved to be an optimized model where it reduces the execution time in comparison to that used by Depth First Search and Breadth First Search. It meets the cyclomatic complexity values when applied to some of the related work examples. It is also proved to be time saving. We have evaluated our proposed approach according to the time required to generate the test cases and the cyclomatic complexity criteria where it proved its efficiency compared to the previous works. In Future, this approach can be enhanced by adding structural diagrams to behavioural ones. Moreover, Test cases can be generated from requirements of different models (waterfall and agile models).

REFERENCES

Boghdady, P.N., N.L. Badr, M. Hashem and M.F. Tolba, 2011a. A proposed test case generation technique based on activity diagrams. Int. J. Eng. Technol., 11: 37-57.

Boghdady, P.N., N.L. Badr, M.A. Hashim and M.F. Tolba, 2011b. An enhanced test case generation technique based on activity diagrams. Proceedings of the 2011 International Conference on Computer Engineering and Systems (ICCES), November 29-December 1, 2011, IEEE, Cairo, Egypt, ISBN: 978-1-4577-0127-6, pp: 289-294.

Chen, L. and Q. Li, 2010. Automated test case generation from use case: A model based approach. Prpceedings of the 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), July 9-11, 2010, IEEE, Heifei, China, ISBN: 978-1-4244-5537-9, pp: 372-377.

Chen, M., P. Mishra and D. Kalita, 2008. Coverage-driven automatic test generation for UML activity diagrams. Proceedings of the 18th ACM Symposium on Great Lakes on VLSI, May 4-6, 2008, ACM, Orlando, Florida, USA, ISBN: 978-1-59593-999-9, pp: 139-142.

Chen, M., P. Mishra and D. Kalita, 2010. Efficient test case generation for validation of UML activity diagrams. Des. Autom. Embedded Syst., 14: 105-130.

Chimisliu, V. and F. Wotawa, 2012. Model based test case generation for distributed embedded systems. Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT), March 19-21, 2012, IEEE, Graz, Austria, ISBN: 978-1-4673-0340-8, pp: 656-661.

Chimisliu, V. and F. Wotawa, 2013. Improving test case generation from uml statecharts by using control, data and communication dependencies. Proceedings of the 2013 13th International Conference on Quality Software, July 29-30, 2013, IEEE, Graz, Austria, ISBN: 978-0-7695-5039-8, pp: 125-134.

Clarke, L.A., 1976. A system to generate test data and symbolically execute programs. IEEE. Trans. Software Eng., 2: 215-222.

Dorigo, M., M. Birattari and T. Stutzle, 2006. Ant colony optimization. IEEE Comput. Intell. Magaz., 1: 28-39.

Doungsa, A.C., K.P. Dahal, M.A. Hossain and T. Suwannasart, 2007. An automatic test data generation from UML state diagram using genetic algorithm. Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007), August 25-31, 2007, IEEE, Cap Esterel, France, pp: 47-52.

Frohlich, P. and J. Link, 2000. Automated Test Case Generation from Dynamic Models. In: Object-Oriented Programming. Elisa, B. (Ed.). Springer, Berlin, Germany, ISBN: 978-3-540-67660-7, pp: 472-491.

Gnesi, S., D. Latella and M. Massink, 2004. Formal test-case generation for UML statecharts. Proceedings of the 9th IEEE International Conference on Engineering Complex Computer Systems, April, 14-16, Florence, Italy, pp: 75-84.

Gutierrez, J.J., M.J. Escalona, M. Mejias and J. Torres, 2006. An approach to generate test cases from use cases. Proceedings of the 6th international Conference on Web Engineering, July 11-14, Palo Alto, California, USA, pp: 113-114.

Gutierrez, J.J., M.J. Escalona, M. Mejias and A.H. Zenteno, 2007a. Using use case scenarios and operational variables for generating test objectives. Syst. Test. Validation, Vol. 23.

Gutierrez, J.J., M.J. Escalona, M. Mejias and J. Torres, 2007b. Derivation of Test Objectives Automatically. In: Advances in Information Systems Development. Wita, W., W.W. Gregory, J. Zupancic, G. Magyar and G. Knapp (Eds.). Springer, Berlin, Germany, ISBN: 978-0-387-70801-0, pp: 435-446.

Harman, M., Y. Jia and W.B. Langdon, 2011. Strong higher order mutation-based test data generation. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, September 5-9, 2011, ACM, Szeged, Hungary, ISBN:978-1-4503-0443-6, pp: 212-222.

Hartmann, J., M. Vieira, H. Foster and A. Ruder, 2004. UML-based test generation and execution. Presentation TAV21 Berlin, 28: 1-5.

Heumann, J., 2001. Generating test cases from use cases. Ration. Edge, Vol. 6,

Iqbal, M.Z., A. Arcuri and L. Briand, 2012. Combining Search-Based and Adaptive Random Testing Strategies for Environment Model-Based Testing of Real-Time Embedded Systems. In: Search Based Software Engineering. Gordon, F. and T.D.S. Jerffeson (Eds.). Springer, Berlin, Germany, ISBN:978-3-642-33118-3, pp: 136-151.

Kansomkeat, S. and W. Rivepiboon, 2013. Automated-generating test case using UML statechart diagrams. Proceedings of the 2003 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology, September 1-3, 2003, South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, ISBN:1-58113-774-5, pp: 296-300.

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.

Korf, R.E., 1985. Depth-first iterative-deepening: An optimal admissible tree search. Artif. Intell., 27: 97-109.

Kulkarni, P. and Y. Joglekar, 2014. Generating and analyzing test cases from software requirements using NLP and Hadoop. Int. J. Curr. Eng. Technol., 4: 3934-3937.

Kundu, D. and D. Samanta, 2009. A novel approach to generate test cases from UML activity diagrams. J. Object Technol., 8: 65-83.

Li, B.L., Z.S. Li, L. Qing and Y.H. Chen, 2007. Test case automate generation from UML sequence diagram and OCL expression. Proceedings of the International Conference on Computational Intelligence and Security, December 15-19, 2007, Harbin, China, pp: 1048-1052.

Li, L., X. Li, T. He and J. Xiong, 2013. Extenics-based test case generation for UML activity diagram. Procedia Comput. Sci., 17: 1186-1193.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Software Eng., 2: 308-320.

McMinn, P., 2004. Search-based software test data generation: A survey. Software Test. Verification Reliab., 14: 105-156.

McMinn, P., 2011. Search-based software testing: Past, present and future. Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), March 21-25, 2011, IEEE, Sheffield, UK, ISBN: 978-1-4577-0019-4, pp: 153-163.

McQuillan, J.A. and J.F. Power, 2005. A survey of UML-based coverage criteria for software testing. Department of Computer Science, Maynooth University, Kildare, Ireland.

Nayak, A. and D. Samanta, 2011. Synthesis of test scenarios using UML activity diagrams. Software Syst. Model., 10: 63-89.

Pechtanun, K. and S. Kansomkeat, 2012. Generation test case from UML activity diagram based on AC grammar. Proceedings of the 2012 International Conference on Computer and Information Science (ICCIS), June 12-14, 2012, IEEE, Songkhla, Thailand, ISBN: 978-1-4673-1937-9, pp: 895-899.

Prasanna, M. and K.R. Chandran, 2011. Automated Test Case Generation for Object Oriented Systems Using UML Object Diagrams. In: High Performance Architecture and Grid Computing. Archana, M., N. Suman, K. Gauravand and K. Sandeep (Eds.). Springer, Berlin, Germany, ISBN:978-3-642-22576-5, pp: 417-423.

Prasanna, M., K.R. Chandran and K. Thiruvenkadam, 2011. Automatic test case generation for uml collaboration diagrams. IETE J. Res., 57: 77-81.

Ray, M., S.S. Barpanda and D.P. Mohapatra, 2009. Test case design using conditioned slicing of activity diagram. Int. J. Recent Trends Eng. IJRTE., 1: 117-120.

Samuel, P., R. Mall and A.K. Bothra, 2008. Automatic test case generation using Unified Modeling Language (UML) state diagrams. IET. Software, 2: 79-93.

Santiago, V., N.L. Vijaykumar, D. Guimaraes, A.S. Amaral and E. Ferreira, 2008. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop ICSTW'08, April 9-11, 2008, IEEE, Sao Jose Dos Campos, Brazil, ISBN: 978-0-7695-3388-9, pp: 63-72.

Sarma, M. and R. Mall, 2007. Automatic test case generation from UML models. Proceedings of 10th International Conference on Information Technology, Dec. 17-20, Washington, USA., pp: 196-201.

Schieferdecker, I., 2012. Model-based testing. IEEE Software, 29: 14-18.

Shahbaz, M., P. McMinn and M. Stevenson, 2015. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. Sci. Comput. Programming, 97: 405-425.

Shanthi, A.V.K. and G.M. Kumar, 2012. Automated test cases generation from UML sequence diagram. Int. Conf. Software Comput. Appl., 41: 83-89.

Shirole, M. and R. Kumar, 2010. A Hybrid Genetic Algorithm Based Test Case Generation using Sequence Diagrams. In: Contemporary Computing. Sanjay, R., A. Banerjee, K.K. Biswas, S. Dua and P. Mishra (Eds.). Springer, Berlin, Germany, ISBN: 978-3-642-14833-0, pp: 53-63.

Shirole, M., A. Suthar and R. Kumar, 2011. Generation of improved test cases from UML state diagram using genetic algorithm. Proceedings of the 4th India Conference on Software Engineering, February 24-27, 2011, ACM, Thiruvananthapuram, Kerala, India, ISBN:978-1-4503-0559-4, pp: 125-134.

Sumalatha, V.M. and G.S.V.P. Raju, 2012. UML based automated test case generation technique using activity-sequence diagram. Int. J. Comput. Sci. Appl. TIJCSA., 1: 58-71.

Sumalatha, V.M. and G.S.V.P. Raju, 2013. Model based test case generation from UML activity diagrams. Int. J. Comput. Sci. Appl. TIJCSA., 1: 46-57.

Swain, R., V. Panthi, P.K. Behera and D.P. Mahapatra, 2012. Test case generation based on state machine diagram. Int. J. Comput. Inf. Syst., 4: 99-124.

Swain, S.K. and D.P. Mohapatra, 2010. Test case generation from behavioral UML models. Int. J. Comput. Appl., 6: 5-11.

Tahbildar, H. and B. Kalita, 2011. Automated software test data generation: Direction of research. Int. J. Comput. Sci. Eng. Surv., 2: 99-120.

Veanes, M., C. Campbell, W. Grieskamp, W. Schulte and N. Tillmann et al., 2008. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In: Formal Methods and Testing. Hierons, R.M., P.B. Jonathan and M. Harman (Eds.). Springer, Berlin, Germany, ISBN: 978-3-540-78916-1, pp: 39-76.

Xing, Y., Y.Z. Gong, Y.W. Wang and X.Z. Zhang, 2015. A hybrid intelligent search algorithm for automatic test data generation. Math. Prob. Eng., Vol. 2015,

Ye, J., Z. Zhan, C. Jin and Q. Zhang, 2009. A Software Test Cases Automated Generation Algorithm Based on Immune Principles. In: Autonomic and Trusted Computing. Jaun, G.N., R. Wolfgang, W. Guojun and J. Indulska (Eds.). Springer, Berlin, Germany, ISBN: 978-3-642-02703-1, pp: 62-74.