Copyright © 2019 Tech Science Press CMES, vol.119, no.3, pp.409-425, 2019

An Automated Approach to Generate Test Cases From Use Case Description Model

Thamer A. Alrawashed[1, *], Ammar Almomani[2], Ahmad Althunibat[1] and Abdelfatah Tamimi[1]

Abstract: Test complexity and test adequacy are frequently raised by software developers and testing agents. However, there is little research works at this aspect on specification-based testing at the use case description level. Thus, this research proposes an automatic test cases generator approach to reduce the test complexity and to enhance the percentage of test coverage. First, to support the infrastructure for performing automatic, this proposed approach refines the use cases using use case describing template and save it in the text file. Then, the saved file is input to the Algorithm of Control Flow Diagram (ACFD) to convert use case details to a control flow diagram. After that, the Proposed Tool of Generating Test Paths (PTGTP) is used to generate test cases from the control flow diagram. Finally, the genetic algorithm associated with transition coverage is adapted to optimize and evaluate the adequacy of such test cases. A money withdrawal use case in the ATM system is used as an ongoing case study. Preliminary results show that the generated test cases achieve high coverage with an optimal test case. This automatic test case generation approach is effective and efficient. Therefore, it could promote to use other test case coverage criteria.

Keywords: Software testing, test cases, software specifications, genetic algorithm.

1 Introduction

Software testing is considered an essential activity of the software development process. The main goal of this activity is to produce high-quality software by executing the software with a good test case to detect bugs, faults, and failure. However, the process of generating such test cases is considered as a complex process, because of the effort, time, and the cost of creating and validating a large number of test cases that require testing software. Other obstacles faced the software testing, are the accuracy of software, which requires a large number of test cases and the adequacy of such test cases [Septian, Alianto and Gaol (2001); Kalaee and Rafe (2016)]. Therefore, effective and efficient testing approaches (e.g., automatic approaches) have an essential role in generating test cases to save time, effort, and cost and to provide more accurate results than manual testing ways that are affected by human errors in which improving the quality of test data is done.

[1] Department of Software Engineering, Alzaytoonah University of Jordan, Airport Street, Amman, Jordan.

[2] IT-department, Al-Huson University College, Al-Balqa Applied University, P. O. Box, Salt, Jordan.

[*] Corresponding Author: Thamer A. Alrawashed. Email: thamer.r@zuj.edu.jo.

Test cases could be generated from the source code (code-based test generation). The test criterion is applied to the source code to produce test requirements, e.g., if the criterion of state testing is applied, the test cases must cover each statement in the source code. The formal and informal specification, which already used to write the source code, is another source for the test cases generation (specification-based test generation). Interestingly, the coverage criteria reveal that execution of the test T on the program 'P' produces the observed behavior (actual output), which should later compared with the expected behavior (expected output). The expected behavior is created based on some knowledge from the specification [Offutt, Xiong and Liu (1999)]. Substantially, the specification is used to produce the source code and to conduct software testing.

The significance of the specification-based test is that the test cases could be produced earlier in the life cycle of software development. Thus, the test is executed before the software development finishes, allowing the software engineer to find the consistencies and ambiguities in the software specifications. Therefore, the specifications could be improved before the source code writing.

The specification-based testing provides many advantages over the code-based testing including the specifications could be used as a guide to design the software testing, reducing the cost of software testing so the cost of software development, and helping the software engineers to discover the problems with the specifications themselves. The specification-based testing also saves the time and resources, allowing the software testing to be performed in the earlier stage of the development life cycle [Ribeiro, Pereira, Rettberg et al. (2018)].

The Unified Modeling Language (UML) is considered the most common specifications standard for software specifications and design. Therefore, many research works concentrate on generating test cases from the UML [Ribeiro, Pereira, Rettberg et al. (2018); Shanthi and MohanKumar (2012); Meziane, Athanasakis and Ananiadou (2008)]. However, the UML is not sufficient to specify the complex and strict software specification. It offers its components (e.g., use case diagram) as graphical diagrams that are informal and could variously be explicated, so other textual language needed to describe such components [Satpathy, Harrison, Snook et al. (2001); Schlauderer and Overhage (2018); Kaczmarek-Heß and de Kinderen (2017); van Eck, Sidorova and van der Aalst (2018)]. In this respect, various textual templates have been developed for software specifications especially to structure the use case description, e.g., use case description template in the Rational Unified Process (RUP), and Cockburn's use case description template [Booch, Jacobson and Rumbaugh (1999); Cockburn (2000); Somé (2009)]. One of the advantages of these templates that supports the automated generation of other behavior models for example activity diagram, sequence diagram, statechart, as well as test cases [Somé (2009)].

Many studies try to generate the test cases from the software specifications [Ribeiro, Pereira, Rettberg et al. (2018); Khurana, Chhillar and Chhillar (2016); Teixeira and e Silva (2017); Shanthi and MohanKumar (2012)]. However, there is a lack of studies to generate the test cases from a formalized model of use cases which is suited to early requirements phases; to apply the acceptable software test coverage criteria in which the adequacy and reliability of such test cases are measured; and to provide a fully automated and detailed test cases generation approach.

For this end, the primary goal of this research work is to generate test cases from the software specifications. An immediate goal is to develop a mechanistic procedure to generate the test cases from use case description model. While the long-term goals are to develop an automatic tool to generate and optimize test cases from the use case description model based on the testing coverage criteria (transition coverage criteria) and a heuristic search algorithm (genetic algorithm). This approach not only provides software developers and test agents with an effective and efficient test generation technique but also with an effective regression test technique, since the highest priority (optimal) test cases are executed earlier in the regression test process than lower priority test cases. Additionally, the proposed approach will provide researchers with a tool that could be used in their future works to generate test cases and prioritization autumnally.

The rest of the study is organized as follows: section two presents the related works. In section three, the proposed approach is addressed. Section four explains the experimentation and research's results. Section five presents an evaluation of the proposed approach. Finally, conclusion and future work are presented in section six.

2 Related work

Contemporary research works had been done to propose tools or techniques to generate test cases from software specifications based on the UML [Teixeira and e Silva (2017); Kalaee and Rafe (2016); Khurana, Chhillar and Chhillar (2016); Mohalik, Gadkari, Yeolekar et al. (2014); Bazi, El Khoury and Srour (2017)].

Tripathy et al. [Tripathy and Mitra (2013)] have proposed an approach to generate test cases from UML sequence and activity diagrams. Both diagrams are converted to the sequence graph and activity graph respectively and then the graphs integrated into one graph known as the system graph. The system graph is converted to the form of test cases and then traversed by the first depth search algorithm.

Khurana et al. [Khurana, Chhillar and Chhillar (2016)] presented an approach to generate the test cases from a use case, sequence, and activity diagram. Each diagram was converted to the graph and then all graphs integrated into a system graph. Similar to the work of (Abinash) the system graph was mutated to the test cases. Although this study used the genetic algorithm to optimize generated test cases, the diagrams, which used to generate the test cases, do not provide enough details like the use case description. Additionally, the study has not automated the process of test cases generation; and has not evaluated the proposed approach.

Sarma et al. [Sarma, Kundu and Mall (2007)] proposed an approach to generate the test cases from the sequence, use case, and sequence diagrams. These diagrams mutated to the system graph. However, the mutation process was not mentioned and the generated test cases were not optimized. Teixeira et al. [Teixeira and e Silva (2017)] presented an automatic approach to generate test cases from an activity diagram. This proposed approach aims to integrate modeling, coding, and testing stage. One of the weaknesses of this study, the activity diagram that was used to generate the test cases does not provide enough details to generate such test cases. The generated test cases also have not been optimized and evaluated.

Another work presents an approach to generate the test cases from the combination of state chart and sequence diagrams. To optimize the generated test cases, the authors used a hybrid bee colony algorithm [Sahoo, Nanda, Mohapatra et al. (2017)]. This study also does not provide a clear automatic approach to generate test cases and does not offer any method to evaluate the proposed approach.

All research works have been used in the UML diagrams to generate the test cases. This comes on the opposite of conclusion that, UML diagrams do not provide enough information to describe a phenomenon, so the proprietary textual notations used. In this sense, the textual notations are used to express the control flow and exceptions, which dependable for all types of white box testing [Riebisch, Philippow and Götze (2002)]. Additionally, as far as, there is no research work provides a fully automatic and evaluated approach to generate the test cases from UML models or to optimize the test cases that automatically generated from UML models. Finally, the previous research works proposed general tools and methods that are not associated with acceptable software test coverage criteria. Thus, this study proposed an automatic approach to generate and optimize the test case from the use case description model using the generic algorithm. This approach is associated with transition coverage criteria, to evaluate the adequacy of generated test cases.

3 Proposed approach

This section proposes an automatic approach to generate and optimize the test cases from the model of use case description using a genetic algorithm and transition coverage criteria. This approach includes four stages including 1) refining use cases 2) automatically converting the use case description to the control flow diagram 3) automatically generating the test cases from the control flow 4) optimizing and evaluating the adequacy of generated test cases by adopting the genetic algorithm to be associated with transition coverage criteria. Fig. 1 presents the proposed approach of this research work.

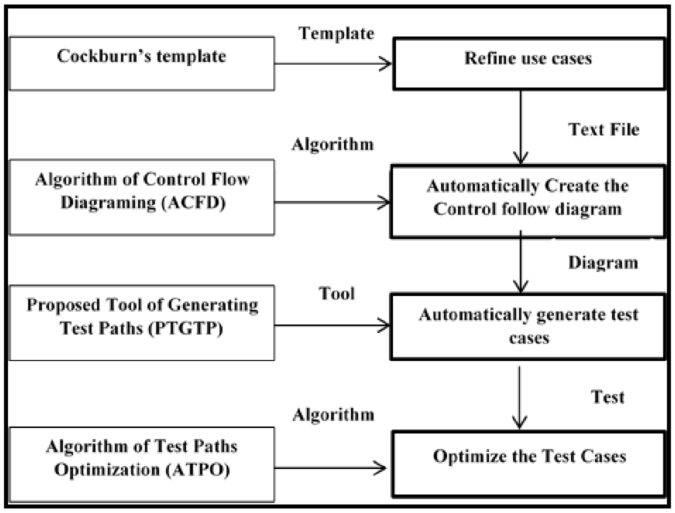

Cockburn's template
Template
Refine use cases
Text File
Algorithm
Algorithm of Control Flow Diagraming (ACFD)
Automatically Create the Control follow diagram
Diagram
Proposed Tool of Generating Test Paths (PTGTP)
Tool
Automatically generate test cases
Test
Algorithm
Algorithm of Test Paths Optimization (ATPO)
Optimize the Test Cases

Figure 1: An approach to generate test cases from the use case description model (AGTCUCM)

3.1 Stage 1: refining use cases

The use case is defined as a simplified overview of the required software functionality. Beyond simplicity, the use cases usually described as a diagram. However, this diagram helps to give an overview of the use case, but the only secondary in importance to the textual description [Riebisch, Philippow and Götze (2002)] which provides a little information of the actual use case. Thus, in order to obtain a deep understanding of such use cases, they need to be refined textually.

Cockburn [Cockburn (1999)] proposed a well-suited template for textual refining use cases. This template aims to create a complete description of a use case by involving all the relevant details including name, goal, actors, pre-conditions, post-conditions, invariant, main success scenario, variations, extensions, and included use cases. Tab. 1 presents this textual notation.

Table 1: Use case description model

Name	use case's title.
Goal	providing the main goal of the use case
Actors	stakeholders who perform the use case.
Preconditions	the system states to be achieved before the use case could be performed.
Postconditions	the system's states could be in after "the use case performed.
Invariants	the system state that holed when and during the course of the use case.
Main Success Scenario	Characterize the use case's objectives could be met as a set of action steps. Staring with the step triggering the use case.
Variations	An alternate set of action steps differ from extensions steps. It is such normal parameters for the use case. Variations could be assigned by reference to the particular step's number in the narrative of the main scenario and referring the step with one or more alternative steps. Unless otherwise expressly stated, the variations change the questions and continue steps in the main scenario to the following steps. Nested variations are assigned by further sub-references.
Extensions	Use cases extend to describe how and when the system response to the exceptional circumstances
Included use cases	Describing a set of use cases that are required to show the behavior of the included use case

Source: [Cockburn (1999)]

The template details are filled in for each use case in software. Such details especially the details in both sections Main Success Scenario and Extensions are very useful for defining the details of the test approaches such as test input and observable behavior. These details will be saved in a text file. To convert the details to the control flow diagram, the text file will be used as input to the Algorithm of Control Flow Diagram (ACFD). In order to write a solid main success scenario, a software engineer should break it down into a set of numbered steps [Cockburn (1999)]. That makes it much easier to be converted to the control flow.

3.2 Stage 2: automatically converting the use cases to the control flow diagram

The main objective of this stage is to propose an algorithm and its tool to automatically create the control flow diagram from relevant details of a use case that represented in the main success scenario and extension sections. These details consist of all scenarios of such a use case and the dependency of each activity. The proposed algorithm (ACFD) represents each step (statement) in the scenario with a node (N) and then uses Edges (E) to connect such nodes to create the control flow diagram as a binary tree. Since each node considers as a parent of the next node and the current node as a child of the previous node. The node could have two children if it represents a conditional statement. Each node comprises a label with textual stereotype represents each step in the use case scenario. The algorithm assigns a weight value for each edge to be used later in the adapted genetic algorithm to optimize and prioritize the generated test cases see Fig. 3 and Example 1.

It is worth mentioning that, the scenario steps correspond to the source code statements such as input, output, and processing statements [Offutt, Xiong and Liu (1999)]. Once the step consists of a conditional statement, the proposed tool splits the statement (step) into two or more statements. The first statement involves the condition itself; the second one involves the statement that runs if a logical expression of the condition is true, and the third one includes the statement that runs if a logical expression of the condition is false. In this case, the proposed algorithm creates a parent node with two children nodes. The parent node represents the condition statement and the children present true and false statements.

Example 1

Suppose that the main success scenario of use case description model includes the following scenario as numbered bullets 1) Input two integers 2) Calculate the summation of these integers 3) And then display the result. These bullets are converted to the control flow as in Fig. 2.

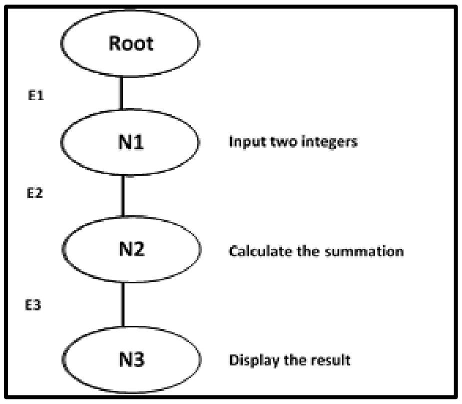
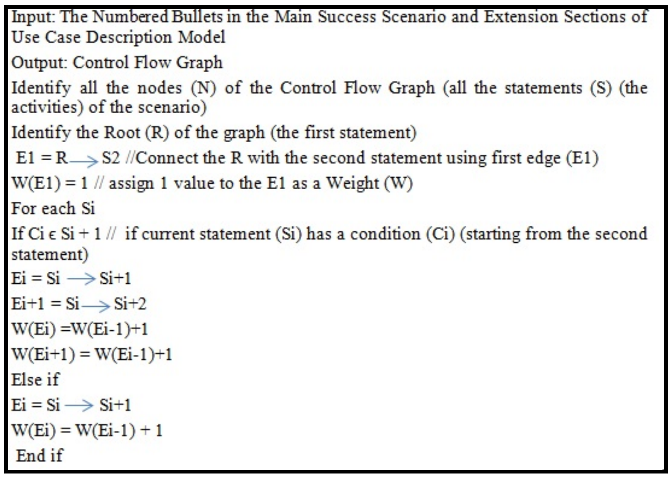


Figure 2: Control Flow Diagram, for a program with a set of nodes N and set of Edges E

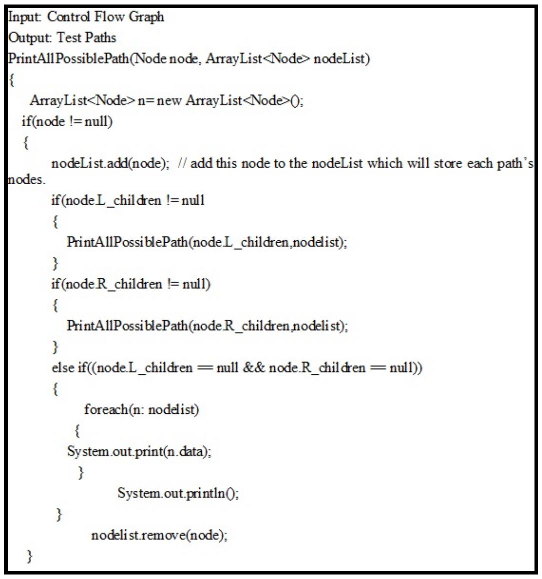


```
Input: The Numbered Bullets in the Main Success Scenario and Extension Sections of Use Case Description Model
Output: Control Flow Graph
Identify all the nodes (N) of the Control Flow Graph (all the statements (S) (the activities) of the scenario)
Identify the Root (R) of the graph (the first statement)
 E1 = R ⟶ S2 //Connect the R with the second statement using first edge (E1)
W(E1) = 1 // assign 1 value to the E1 as a Weight (W)
For each Si
If Ci ϵ Si + 1 //  if current statement (Si) has a condition (Ci) (starting from the second statement)
Ei = Si ⟶ Si+1
Ei+1 = Si ⟶ Si+2
W(Ei) =W(Ei-1)+1
W(Ei+1) = W(Ei-1)+1
Else if
Ei = Si ⟶ Si+1
W(Ei) = W(Ei-1) + 1
 End if
```

Figure 3: High-level description of the Algorithm of Control Flow Diagram (ACFD), for a program with a set of Statements S and Conditions C; set of Nodes N; set of Edges E and Weight W for each edge

3.3 Stage 3: automatically generating test cases from the control flow graph

This stage aims to propose a tool to generate the test paths from the control flow diagram, where the final set of the test cases result from test paths. The Proposed Tool *of Generating Test Paths* (PTGTP) declares a method to distinguish each path in the graph. This method receives two parameters including the node where the path starts from and an array list. If this node is not null, it will be added to the array list. Then the method will be invoked recursively for left and right nodes (children) until reaching the end of a path it will be stored in an array list of lists (paths) see Fig. 4.



```
Input: Control Flow Graph
Output: Test Paths
PrintAllPossiblePath(Node node, ArrayList<Node> nodeList)
{
   ArrayList<Node> n= new ArrayList<Node>();
  if(node != null)
  {
      nodeList.add(node);  // add this node to the nodeList which will store each path's nodes.
      if(node.L_children != null
      {
         PrintAllPossiblePath(node.L_children,nodelist);
      }
      if(node.R_children != null)
      {
         PrintAllPossiblePath(node.R_children,nodelist);
      }
      else if((node.L_children == null && node.R_children == null))
      {
           foreach(n: nodelist)
          {
         System.out.print(n.data);
           }
                  System.out.println();
      }
            nodelist.remove(node);
  }
```

Figure 4: Proposed Tool of Generating Test Paths (PTGTP)

3.4 Stage 4: optimizing the test cases

The strategy which applied to generate the test paths, influence directly the quality of test cases. In this respect, the test cases should be adequate to test the software. Thus testing agents should use adequacy criteria to decide whether the systems have adequately tested for a specific test criterion [Frankl and Weyuker (1988)]. The adequacy is evaluated based on coverage that defined by many criteria. These criteria have been categorized into code-based testing criteria, specification-based testing criteria, and model-based testing criteria. Definitely, the proposed approach adapts the model-based testing which is an effective combination between of code-based testing and specification-based testing. The coverage criteria of this software testing include four levels: the transition coverage level, the full predicate coverage level, the transition-pair coverage level, and the complete sequence level. All levels could be used, but do a benefit/cost tradeoff one of these levels could be applied [Offutt, Xiong and Liu (1999)]. Because of the transition coverage, level requires a fewer test case than other levels especially the full predicate coverage level; it adopted in this research. The transition coverage intends to verify all edges in the control flow graph. Its result is a ratio between the traversed edges and all edges in the graph. This criterion guarantees that each edge in the graph should traverse at least one [Malhotra and Garg (2011)].

One of the proposed approach targets is to optimize and prioritize the test cases by achieving maximum transaction coverage. To do so an evolutionary algorithm (genetic algorithm) should be associated with the transition coverage's criteria.

The Genetic Algorithm (GA) is a heuristic search algorithm used to solve various optimization problems based on the evolutionary ideas of natural selection and genetics [Malhotra and Garg (2011)]. In terms of inputs change, the genetic algorithms are more robust than other Artificial intelligence (AI) systems. They also provide significant benefits in searching large state-space over other optimization techniques [Sumalatha (2013)]. Genetic algorithm finds optimal solutions by applying various phases namely: initialization, evaluation, selection, crossover, and mutation.

3.4.1 Initialization phase

In the initialization phase, the initial population of individuals or chromosomes is generated. The individuals are a set of input variables' values, which obtained from the input domain. The size of each depends upon the number and range of the input values. These initial values of individuals are generated randomly or seeding depending on the tester knowledge.

3.4.2 Evaluation phase

The fitness value of each individual is calculated in the evaluation phase. The fitness assesses the goodness of each individual relative to the global optimum solution. The individuals with high fitness values are more near the optimum solution over these with fewer fitness values.

3.4.3 Selection phase

Selection phase is replication some individuals from the current population based on their fitness values. This means only the best individuals are transmitted from the current generation to the next generation. The output of this phase is a mating pool that involves the individuals that mate with together to generate the offspring.

3.4.4 Crossover phase

Crossover phase is the process of a random exchange of genetic material between two. Individuals that governed by the crossover site. Two individuals are selected from the mating pool and then mating to obtain the offspring. Consequently, these individuals exchange their string as determined by the crossover site.

3.4.5 Mutation

Mutation refers to a random change of a bit in an individual, for example, flipping 0 to 1 or vice versa [Malhotra and Garg (2011); Holland and Goldberg (1989)].

In order to connect this stage with the stage three, after running the proposed tool in stage three and generate the test paths, the Algorithm of Test Paths Optimization (ATPO) Fig. 5 takes all such test paths as the first population. Each test path in this population represents a chromosome. The fitness value for each chromosome is calculated using Eq. (1), where the x=number of transitions (number of edges) in each path (chromosome) and then the probability of each chromosome calculated using Eq. (2). This algorithm also involves the fitness calculation module which responsible for randomly generating the initial population of chromosomes, receiving the fitness value (comparing with the sum of edges in the graph) of each chromosome from the previous module, running crossover and mutation operations to produce the new population. The steps of this algorithm repeated until a path that achieves maximum transition coverage is detected.

```
Input: All Test Paths from ATPG
Output: Optimized Test Cases
Calculate the fitness value for each path using fitness equation 1.
          $F(x)= x * x$          (1)
Calculate the probability of each chromosome using equation 2.
          $P(i)= F(x)/\sum F(x)$      (2).
Generate random numbers to create a new population.
The crossover operation has been used to mate the pairs of individuals. Employ single
point crossover from the third bit from the right.
The mutation equation is applied to mutate every fifth bit if the generated random
number is less than 0.5 [7]
All the steps from (1-5) are repeated until a path that achieves maximum transitions is
detected.
End
```

Figure 5: High level of description of an Algorithm of Test Paths Optimization (ATPO), for a problem withthe function of fitness calculation F(x); the population of each path x; the function of probability calculation P(i)

4 Experimentation

4.1 Case studies

In order to demonstrate the feasibility and efficiency of the proposed automatic approach, three small use cases as case studies have been adapted including: the ATM system especially money withdrawal use case description used as an ongoing example in this paper, a file transfer use case in the File Transfer Protocol (FTP) system, and enter meeting use case in the virtual meeting system [Nebut, Fleurey, Le Traon et al. (2006)].

The file transfer use case includes five sequential steps associated with one exception step, and the enter meeting use case consists of four sequential steps. Where The main scenario of the money withdrawal use case consists of approximately thirteen sequential steps associated with seven exception steps as shown in Tab. 2.

Table 2: Use case description model of ATM withdraw money

Name	Withdraw Money
Goal	Allow authorized user to withdraw a limited amount of money
Actors	Bank customers (ATM Cardholders) and bank staff
Preconditions	The bank customer has a valid ATM card; ATM has cash money
Postconditions	Receipt printed
Invariants	None
Main Success Scenario	A bank customer inserts debt cards and enters a PIN If the PIN is Valid Then ATM displays available actions the customer selects withdraw cash from available actions ATM promotes Account the customer selects an account ATM promotes amount the customer enters the desired amount IF the customer has sufficient funds Then IF the desired amount is within the allowable limit Then IF the ATM has sufficient cash Then money is dispensed and receipt is printed
Variations	None
Extensions	2a. Else ATM displays an error message 9a. ATM prints error message 9b. and asks the customer to re-enter the amount 10a. shows allowable limit 10b. asks the customer to re-enter the cash amount Else

	11a. Else ATM technician is alerted and 11b. ATM displays Error message Else ATM displays an error message
Included use cases	Balance inquiry

The numbered steps of the success main section and extensions section of the previous template model are saved as a text file to be used later as input to the ACFD. As explained in stage two of the proposed approach, the tool of ACFD converts the steps (statements) in the text file into the control flow graph. A node represents each main scenario and exception step. Later, the edges are used to connect these nodes as a binary tree (control flow graph). Once the step (statement) has a condition, the tool splits this statement into three statements the first one for the condition, the second one for the event that executed if the condition is true and the third one for the event that executed if the condition is false. Fig. 6 presents the control flow graph of withdrawing money use case in the ATM system, which automatically results from the proposed tool of ACFD.

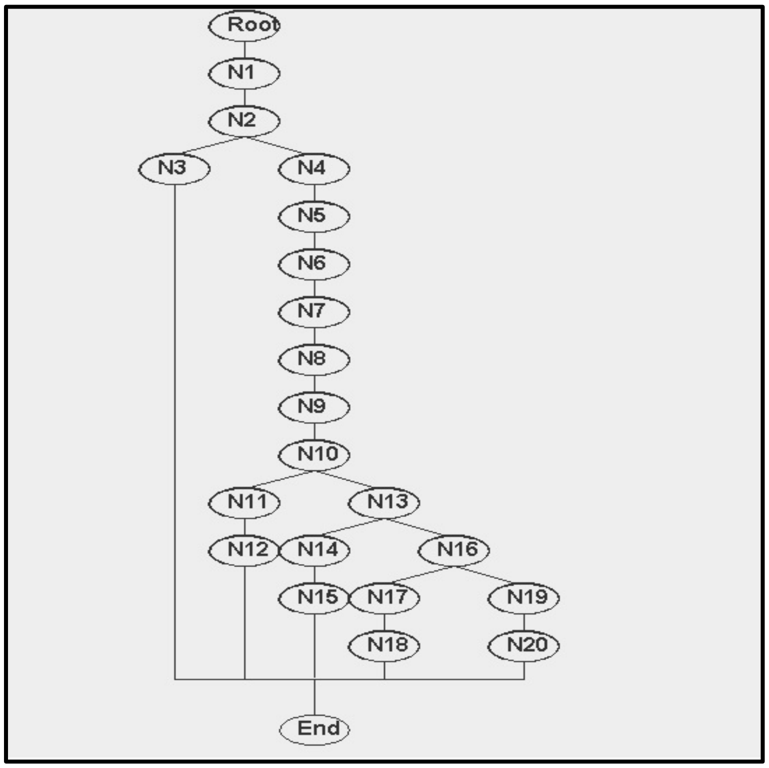
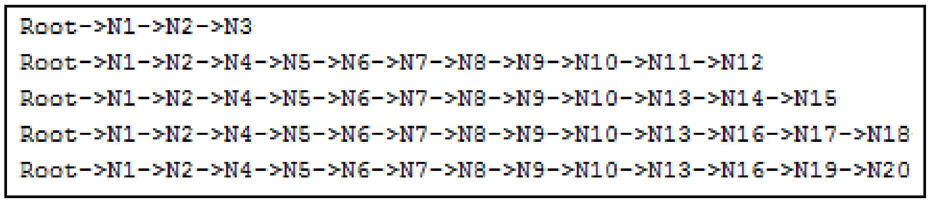


Figure 6: control flow graph of withdrawing money use case in the ATM system

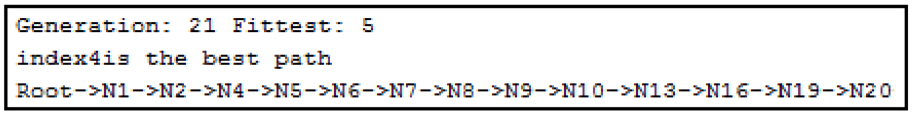
Consequently, this control graph of money withdrawal use case's scenario was used as input to the Proposed Tool of Generating Test Paths (PTGTP), which outputs the test paths. After running the PTGTP, five test paths have been generated see Fig. 7.



```
Root->N1->N2->N3
Root->N1->N2->N4->N5->N6->N7->N8->N9->N10->N11->N12
Root->N1->N2->N4->N5->N6->N7->N8->N9->N10->N13->N14->N15
Root->N1->N2->N4->N5->N6->N7->N8->N9->N10->N13->N16->N17->N18
Root->N1->N2->N4->N5->N6->N7->N8->N9->N10->N13->N16->N19->N20
```

Figure 7: Generated test paths

In order to optimize and evaluate the adequacy of the test paths, the proposed tool of Algorithm 2 (ATPO) has been executed with the generated test paths from PTGTP as inputs. Fig. 8 shows after 21 generations have been generated, the test path number 4 found as an optimized test path (test case). This result reveals that test path number 4 is the best path to cover maximum transactions in the control flow graph of the money withdrawal use case.



```
Generation: 21 Fittest: 5
index4is the best path
Root->N1->N2->N4->N5->N6->N7->N8->N9->N10->N13->N16->N19->N20
```

Figure 8: Optimized test-path

Due to the space constraints, explanation of applying the proposed approach on the money withdrawal use case is only provided. However, Tab. 3 presents the statistics on the generated test cases, the optimized test case, and the efficiency of the optimized test case in term of transition coverage form three case studies.

Table 3: Statistics on the generated test cases

Number	Use case	Generated test cases	Index of optimized test case	Percentage of transition coverage
1	ATM cash Withrawal	5	4	65%
2	Virtual meeting	1	0	100%
3	File transfer protocol	2	1	57%

As shown in Tab. 3, five test cases are generated from the money withdrawal uses case.

The optimized one among these test cases is the test case with the index four as also shown in Fig. 8 since it covers 65% of the transitions in the flow graph of the use case's statements, while the number of covered transitions is a very important measure of test efficiency. This measure is widely applied in industry and scientific research when a rigorous mapping from requirements to code is mandatory [Nebut, Fleurey, Le Traon et al. (2006)]. To reach 100% coverage of transition, test agents should use all the test cases. Regarding the enter meeting use case, one test case has been generated from this use case and this test case covers 100% of the transitions. Finally, two test cases have been generated from file transfer use case and the second test case covers most of the transitions 57%.

Consequently, the test agents can use this approach to determine the test cases which required to conduct an adequate test and which one of these test cases has a high impact on software testing.

Important contributions for the proposed approach from an approach was proposed by Nebut et al. [Nebut, Fleurey, Le Traon et al. (2006)], where the authors described a complete testing approach to generate test cases from requirements contracts. Interestingly, the proposed approach in this paper depends on the graphical notation (use case description model) which provides adequate details on the software specifications than the contract of requirements which used in Nebut's approach. Not only the main statements of the use case's specifications but also the exception statements are intensively mentioned in the use case description model-such a statement represented as main nodes in the graph diagram and test paths. Thus, the robustness criterion results in the proposed model are encouraged. This is on the contrary to the Nebut's approach where the robustness criterion results were disappointed [Nebut, Fleurey, Le Traon et al. (2006)].

In term of time-consuming the proposed approach more efficient than the Nebut's approach. In the proposed approach, the test agents do not need to analyze the requirements contract as in the Nebut's approach to determining the use cases, since the requirement specifications already analyzed by the system analyst. Another contribution of the proposed approach than Nebut approach is generating not only a set of test cases to achieve 100% transition coverage, but also providing the test agent with the best test case that achieves highest transition coverage. Finally, the Nebut's approach applied the coverage of regular expressions that lead to a large number of test cases, on the contrary to this research proposed approach which based on the coverage of transition that leads to the exact number of the test cases required to cover the transition between the statements of use cases.

5 Evaluation

5.1 Test coverage

The proposed approach generates the test cases to increase coverage on different types of software considering the transactions between the nodes that represent the software specifications. As test paths generated by the proposed approach execute a maximum number of software specification's statements by means of the test case covers the maximum number of transactions of each specification.

5.2 Cyclomatic complexity test

This section discusses the validity of the test paths. The validity of the test paths had been evaluated by a Cyclomatic complexity. The Cyclomatic complexity is a software metric used to measure the software complexity [McCabe (1976)]. Thomas developed this metric and McCabe in 1976 based on a software's control flow graph, which involves a set of nodes, and edges, see Eq. (3).

$$V(G)= E - N + 2 \tag{3}$$

Where V(G) is the Cyclomatic complexity; E is the number of edges; and N is the number of nodes in the control flow diagram.

In the term of software testing, there is a strategy known as basis path testing, uses the Cyclomatic complexity metric to test each linearly independent path through the control graph, where the number of test paths equals the software's Cyclomatic complexity value [McCabe (1976)]. In this research, by applying such a metric for the generated test paths, as shown in Fig. 2.

$E= 25, N= 22.$

$V (g) = 25- 22 + 2 = 5$

Therefore, the Cyclomatic complexity value is 5 and the number of generated test paths is 5. This result reveals that the generated test cases from the proposed approach will be compatible with the complexity of software that will represent the case of this research (money withdrawal) and the cost and effort required for testing will be less [McCabe (1976)].

6 Conclusion

Although many research, works have been done to propose approaches to generate test cases from UML models (design models), no research works conducted to generate test cases from use case description model, which provides many details relevant to the test cases.

Additionally, when they developed approaches-based testing, the previous studies had not considered the control flow and test adequacy, which are important for any qualified test approach. Thus, this study proposed a fully automatic test approach to generate time-efficient test cases from the use case description model for high-confidence software not only to detect the faults as much as possible in the early stage of software development but also to examine the validity of software requirements. This proposed approach considers the use case description of each function, rules of mapping it to the control flow diagram, rules of generating test cases from this control flow diagram to quickly generate the test cases with high coverage on different types of complex information systems. The mapping rules and its tool have been proposed to convert the use case description to the control flow diagram, where the generating rules and its tool have been proposed to generate the test cases (test paths) from the control flow diagram. Furthermore, the researchers adapted the genetic algorithm to optimize and evaluate the adequacy of generated test cases.

The experiments show that the proposed approach is efficient and effective in providing the software tester with a near-optimal test case and high test coverage in the early stage of software development, which cannot be done by contemporary similar studies [Teixeira and e Silva (2017); Khurana, Chhillar and Chhillar (2016); Mohalik, Gadkari, Yeolekar et al. (2014)]. Interestingly, several studies have been conducted to generate the test cases from the software specifications [Binder (2000); Legeard, Peureux and Utting (2002); Malhotra and Garg (2011); Nebut, Fleurey, Le Traon et al. (2006)]. However, there are many important points to distinguish this study from such research works. First, this study generates the test cases from use case formalized model in contrast with other studies which depend on the formal methods, where the requirements are written in a formal language [Binder (2000); Legeard, Peureux and Utting (2002); Meziane, Athanasakis and Ananiadou (2008)]. Second, the proposed approach in this study is based on a specific test adequacy criterion in which generating the smallest number of efficient test cases, where other studies such as Briand and Labiche [Briand and Labiche (2002)] used regular expressions that lead to a large number of test cases. Third, this study differs from other research works such as Nebut et al. [Nebut, Fleurey, Le Traon et al. (2006)] by providing not only a set of test cases to achieve 100% transition coverage, but also providing the test agent with the best test case that achieve highest transition coverage. Thus, it could be a promising approach for the testing techniques that use transition coverage of the control flow graph. The results of the proposed approach are encouraging. It successfully generates the test cases that can achieve the highest test coverage with a near-optimal number of test cases. However, further studies could be conducted to continuous in converting the software specifications (use case description) to code and measure the statement coverage to extracting the relationship between the transition coverage and statement coverage criteria. Furthermore, this automatic test cases generation approach could also be used in the future work with other test coverage criteria.

References

Bazi, G.; El Khoury, J.; Srour F. J. (2017): Integrating data collection optimization into pavement management systems. *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 135-146.

Binder, R. (2000): *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional.

Booch, G.; Jacobson, I.; Rumbaugh, J. (1999): *The Unified Software Development Process*. Reading Addison Wesley.

Briand, L.; Labiche, Y. (2002): A UML-based approach to system testing. *Software and Systems Modeling*, vol. 1, no. 1, pp. 10-42.

Cockburn, A. (1999): Structuring use cases with goals. *Journal of Object-Oriented Programming*, vol. 35, no. 40, pp. 56-62.

Cockburn, A. (2000): *Writing Effective Use Cases, the Crystal Collection for Software Professionals*. Addison-Wesley Professional Reading.

Frankl, P. G.; Weyuker, E. J. (1988): An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483-1498.

Holland, J. H.; Goldberg, D. (1989): *Genetic Algorithms in Search, Optimization and Machine Learning*. Massachusetts: Addison-Wesley.

Kaczmarek-Heß, M.; de Kinderen, S. (2017): A multilevel model of IT platforms for the needs of enterprise IT landscape analyses. *Business & Information Systems Engineering*, vol. 59, no. 5, pp. 315-329.

Kalaee, A.; Rafe, V. (2016): An optimal solution for test case generation using ROBDD graph and PSO algorithm. *Quality and Reliability Engineering International*, vol. 32, no. 7, pp. 2263-2279.

Khurana, N.; Chhillar, R. S.; Chhillar, U. A. (2016): A novel technique for generation and optimization of test cases using use case, sequence, activity diagram and genetic algorithm. *Journal of Software*, vol. 11, no. 3, pp. 242-250.

Legeard, B.; Peureux, F.; Utting, M. (2002): Automated boundary testing from Z and B. *International Symposium of Formal Methods Europe*, pp. 21-40.

Malhotra, R.; Garg, M. (2011): An adequacy based test data generation technique using genetic algorithms. *Journal of Information Processing Systems*, vol. 7, no. 2, pp. 363-384.

McCabe, T. J. A. (1976): Complexity measure. *IEEE Transactions on Software Engineering*, vol. 4, no. 1, pp. 308-320.

Meziane, F.; Athanasakis, N.; Ananiadou, S. (2008): Generating natural language specifications from UML class diagrams. *Requirements Engineering*, vol. 13, no. 1, pp. 1-18.

Mohalik, S.; Gadkari, A. A.; Yeolekar, A. (2014): Automatic test case generation from Simulink/Stateflow models using model checking. *Software Testing, Verification, and Reliability*, vol. 24, no. 2, pp. 155-180.

Nebut, C.; Fleurey, F.; Le Traon, Y.; Jezequel, J. M. (2006): Automatic test generation: a use case driven approach. *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140-55.

Offutt, A. J.; Xiong, Y.; Liu, S. (1999): Criteria for generating specification-based tests. *ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems*, pp. 119-129.

Ribeiro, F. G. C.; Pereira, C. E.; Rettberg, A.; Soares, M. S. (2018): Model-based requirements specification of real-time systems with UML, SysML, and MARTE. *Software & Systems Modeling*, vol. 17, no. 1, pp. 343-361.

Riebisch, M.; Philippow, I.; Götze, M. (2002): UML-based statistical test case generation. *International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pp. 394-411.

Sahoo, R. K.; Nanda, S. K.; Mohapatra, D. P.; Patra, M. R. (2017): Model driven test case optimization of UML combinational diagrams using hybrid bee colony algorithm. *International Journal of Intelligent Systems and Applications*, vol. 9, no. 6, pp. 43-54.

Sarma, M.; Kundu, D.; Mall, R. (2007): Automatic test case generation from UML sequence diagram. *International Conference on advanced Computing and Communications*, pp. 60-67.

Satpathy, M.; Harrison, R.; Snook, C.; Butler, M. (2001): A comparative study of formal and informal specifications through an industrial case study. *IEEE/IFIP Workshop on Formal Specification of Computer Based Systems* .

Schlauderer, S.; Overhage S. (2018): BoSDL: an approach to describe the business logic of software services in domain-specific terms. *Business & Information Systems Engineering*, vol. 1, no. 21, pp. 393-413.

Septian, I.; Alianto, R. S.; Gaol, F. L. (2001): Automated test case generation from UML activity diagram and sequence diagram using depth first search algorithm. *Procedia Computer Science*, vol. 116, no. 1, pp. 629-637.

Shanthi, A. V. K.; MohanKumar, G. (2012): A novel approach for automated test path generation using TABU search algorithm. *International Journal of Computer Applications*, vol. 48, no. 13, pp. 28-34.

Somé, S. S. (2009): A meta-model for textual use casedescription. *Journal of Object Technology*, vol. 8, no. 7, pp. 87-106.

Sumalatha, V. M. (2013): Object-oriented test case generation technique using genetic algorithms. *International Journal of Computer Applications*, vol. 61, no. 20, pp. 20-26.

Teixeira, F. A. D.; e Silva, G. B. (2017): EasyTest: an approach for automatic test cases generation from UML activity diagrams. *Information Technology-New Generations*, pp. 411-417.

Tripathy, A.; Mitra, A. (2013): Test case generation using an activity diagram and sequence diagram. *Proceedings of International Conference on Advances in Computing*, pp. 121-129.

van Eck, M. L.; Sidorova, N.; van der Aalst, W. M. (2018): Guided interaction exploration and performance analysis in artifact-centric process models. *IEEE 19th Conference on Business & Information Systems Engineering*, pp. 1-5.