Computer and Information Science; Vol. 8, No. 1; 2015
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Best Test Cases Selection Approach Using Genetic Algorithm

Nidal Yousef[1], Hassan Altarwaneh[1] &Aysh Alhroob[1]

[1] Faculty of Information Tecnolology, Isra University, Amman, Jordan

Correspondence: Nidal Yousef, Faculty of Information Tecnolology, Isra University, Amman, Jordan. E-mail: hassan_trawneh@iu.edu.jo

Received: November 2, 2014 Accepted: November 5, 2014 Online Published: January 2, 2015

doi:10.5539/cis.v8n1p25 URL: http://dx.doi.org/10.5539/cis.v8n1p25

Abstract

This paper proposes an approach for selecting best testing scenarios using Genetic Algorithm. Test cases generation approach uses UML sequence diagrams, class diagrams and Object Constraint Language (OCL) as software specifications sources. There are three main concepts: Edges Relation Table (ERT), test scenarios generation and test cases generation used in this work. The ERT is used to detect edges in sequence diagrams, identifies their relationships based on the information available in sequence diagrams and OCL information. ERT is also used to generate the Testing Scenarios Graph (TSG). The test scenarios generation technique concerns the generation of scenarios from the testable model of the sequence diagram. Path coverage technique is proposed to solve the problem of test scenario generation that controls explosion of paths which arise due to loops and concurrencies. Furthermore, GA used to generates test cases that covers most of message paths and most of combined fragments (loop, par, alt, opt and break), in addition to some structural specifications.

Keywords: test cases, software specifications, UML diagram, genetic algorithm

1. Introduction

There are many forms of testing a software system. One of them is structural specifications testing [1]. On the other hand, testing of behavioral specifications of a software system is very important to evaluate the system interaction and for testing scenarios. A critical component of testing is the construction of test cases. Test cases may be used to detect any faults and problems that exist in the software design even before the program is implemented. A test case contains test input values, expected output and the constraints of preconditions and post conditions for the input values. Generation and selection of the effective test cases from UML models is one of the most challenging tasks [3].

Sequence diagram is one of the main UML design diagrams. It is used to represent the behavioral details and specifications of a software system. However, sequence diagram alone does not express liveness/progress properties or can differentiate between necessary and possible behaviour [2]. To address these limitations, UML class diagram and OCL are also used in this work to model the software and generate test cases more accurately. This paper aims to generate test cases that are able to cover the interaction faults and scenario faults. Figure 1 represents the proposed approach in this work and introduces three main components: Edges Relation Table (ERT), Test Scenarios Generation and Test Cases generation. The scenarios generation technique concerns the generation of scenarios from the testable model of the sequence diagram. Coverage of all paths is a major problem in generating test scenarios as explosion of paths which arise due to loops and concurrencies is to be properly dealt with. Each combined fragment could be one scenario or combined testing scenarios (interaction). Information Table (InfT) is proposed to enrich the testing scenarios with necessary information being extracted from class diagram [1].

This paper is organized as follows: A related work is described in Section 2. Decomposing Sequence Diagram into Edges is described in Section 3. Edge relationship table is discussed in Section 4. Section 5 describes Test cases generation approach. Finally, Section 6 draws the conclusions and proposes future work.

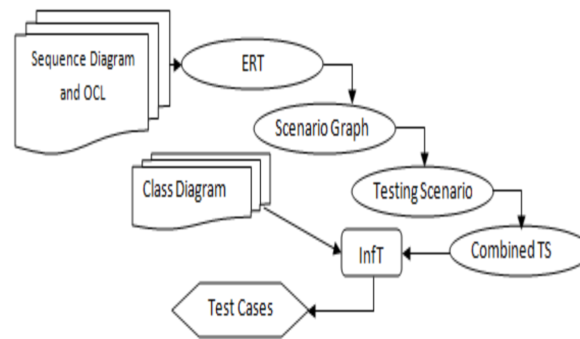

Figure 1. Test cases generation model

2. Previous Work

The previous works provide automatic and manual methodologies to extract test cases from one primary UML diagram. These methods also use some other UML diagrams to obtain additional information. These approaches do not de ne a precise model for the generation of test paths which lead to generation of test cases. Many fault types may exist in different combinations of paths / statements. Detection of such faults in all combinations is a hard problem as the number of statements and faults increases with the size of a software system. Systematic Test cases generation technique is needed to overcome this combinatorial problem [14].

The research in [15] considered reusing of common activities in model based testing and model based development. The authors presented a case study (microwave oven) in which the executable and translatable UML system models were used for automatically generating test models in the QTronic Modeling Language using horizontal model transformations.

Genetic Algorithms are used in [16] to generate and optimize test cases from UML State Chart diagram. Crossover method of GA is applied to generate the test sequence and the Mutation Analysis is used for evaluating the efficiency of the test sequence. The proposed algorithm was tested using a case study of driverless train. The authors concluded that GA can be used for test case generation; however, it can only be used effectively if there are a large number of test cases and large number of possible test sequences. Research in [17] also used GA for improved test case generation. The authors presented an approach that combine information from UML state chart diagram used as finite state machines with GA. Test cases were generated by transforming EFSMs into extended control flow graph. GA is applied to generate feasible test sequence and test data. The work is primarily motivated by three factors. First, state-based testing is widely used in protocol software and embedded system software. Second, generation of test sequences using brute force method is very expensive. Third, generation of test sequences using random test cases may not always lead to feasible test paths, these needs to satisfy the guard condition.

The authors noted that superior results were reported by GA based technique.

The authors in [18] consider the problem of automatically generating test cases for dynamic specification mining that observes program execution to infer models of normal behavior. If a sufficient number of tests are not available, the resulting specification may be too incomplete to be useful, therefor necessitating the requirement for systematic test case generating. The novelty of their approach lies in combining systematic test case generation and type state mining. TAUTOKO, a type state miner is used to generate test cases that cover previously unobserved behavior, systematically extending the execution space, and enriching the specification.

Researchers in [19] considered user-driven test case generation aimed at efficiently testing multimedia applications concerning Digital TV (DTV) receivers and Set-Top Boxes (STB). The proposed approached was experimentally tested for correlation between the detected DTV functional failures using the proposed user-driven test-case generation method and subjectively perceived failures. Improvement in testing efficiency and reduction compared to referent approaches.

The research work in [20] tackled the problem of detecting and improving the test cases after mutation a problem where artificial bugs referred to as mutants are injected into software and the resulting test cases are executed on these fault injected version. They proposed an approach, µTEST that automatically generates unit tests for object-oriented classes based on mutation analysis.

S. Ali et. Al. published a review of the application and empirical investigation of search-based test generation. Readers are referred to [21] for a detailed overview of studies concerning search-based software testing and their empirical evaluation.

3. Decomposing Sequence Diagram Into Edges(Nodes)

The simplicity of sequence diagrams makes it easier for the customers to express the requirements and understand them. Unfortunately, shortage or lack of semantic content in sequence diagrams makes them ambiguous and therefore difficult to interpret [7]. Examination of the informal documentation almost resolves the ambiguity but, in some cases, these ambiguities may go undetected leading to costly software errors. To overcome this problem, two solutions could be used. Firstly, the user may provide messages with complete semantic information. Secondly, if sufficient semantic information is not available, one may interpret sequence diagrams based on some heuristics. In the proposed approach in this paper, a compromise has been done, whereby messages in a sequence diagram may be annotated with a pre/post-condition style specification expressed in OCL as shown in Figure 3. Note that this is only a small additional burden on the user since the amount of information required by our methodology in this paper is actually very small.

The specifications should include the declaration of global state variables where a state variable represents some important aspect of the system, e.g., whether or not the user has inserted valid coin into the coffee machine mentioned in Figure 2. The pre/post-conditions should then include references to these variables.

Now, there are two kinds of constraints on a sequence diagram: constraints on the state vector given by an OCL specification, and the constraints on the ordering of the messages given in the sequence diagram [4]. Edges structure generation is based on the edges vector as shown in Notation 1; edges vector is responsible for merging a proper object with its message. Thus, the sequence diagram edges will be presented as follows:

$$s_0 \to m_1 \to s_0', s_1 \to m_2 \to \cdots \to m_{r-1} \to s_{r-1}', s_r \to m_r \to s_r \quad (1)$$

Where, the m_i is a message between objects and$s_i, \acute{s}_i$, are the edges state vector immediately before and after message m_i is executed. The source and destination object of message mi are denoted by m_i^{source} and m_i^{dest}, respectively. S_i denotes either$s_i$or $\acute{s}_i$.S_i' is the *jth* element of the vectorS_i. Let v_j denote the name of the variable associated with position j in the edge vector.

The initial edges vectors are generated directly from the message specifications: if $m_i$has precondition $v_j = y$ then let $S_i[j] := y$, and if $m_i$has a post condition$v_j = y$, let$S_i[j] := y$.

Identifying edges is not enough to establish a coherent testing scenario. These edges are connected by many types of relationships in a sequence diagram. These relations must be identified to build an integrated testing scenario graph.

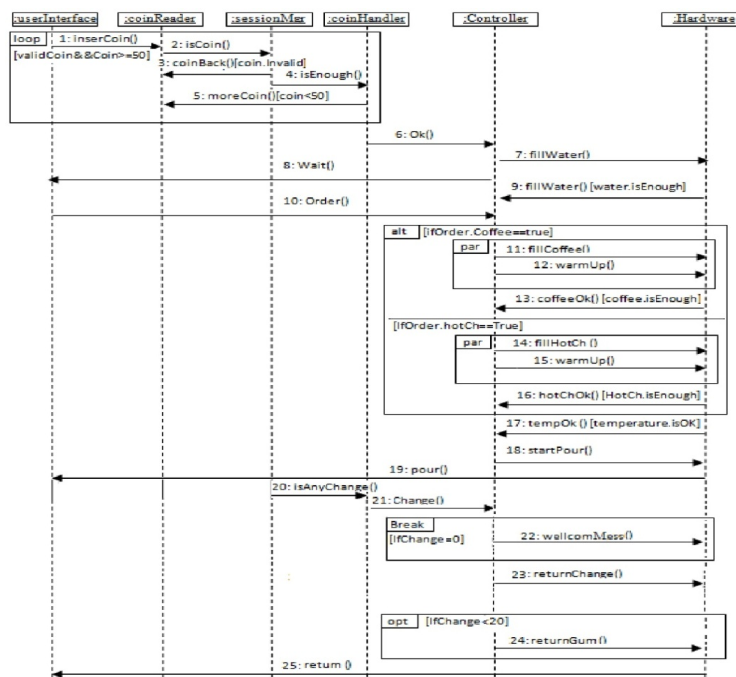

Figure 2. Sequence Diagram for a coffee machine

```
coinIn, coinback, coinmore, fillwater, wait: Boolean
coin: coinamonut, water: Sequence
     Insert coin(c: Coin)
       pre: coinIn = false
       post: coinIn = true and coin = c
     coinBack()
       pre: coinIn = true and coin =false
       post: coinIn = false
     moreCoin()
       pre: coinIn = true and coin = true
       post: coinIn = true
     fillWater(w:Water)
       pre: coinIn = true and coin = true coinAmount=true
       post: wait = true
```

Figure 3. CL style specifications

4. Edges Relationships Table (ERT)

Alhroob, Dahal and Alamgir [1] proposed an automatic approach to generate hierarchy table to set the relationships among classes in a systematic way. In this paper, this hierarchy table is enhanced to the Edges Relationships Table (ERT) to set the relationships between the edges automatically.

4.1 Testing Scenarios Graph (TSG)

In this section, the TSG is defined and then we present the proposed methodology to generate TSG from a sequence diagram. TSG is defined as follows (Notation 2):

$$TSG=\left[N_{TSG},\sum_{TSG} q0_{TSG},F_{TSG},PAR_{TSG},SEL_{TSG},BR_{TSG}\right] \tag{2}$$

Where, $N_{TSG}$is the set of all nodes of testing scenarios; each node basically represents an event.

$\sum_{TSG}$is the set of edges representing transitions from one state to another. $q0_{TSG}$is the initial node representing a state from which an operation begins. F_{TSG} is the set of final nodes representing states where an operation terminates.

$PAR_{TSG}$is the set of parallel edges and it represents states where one operation red other operations at the same time.

$SEL_{TSG}$is the set of either alt or opt edges.

$BR_{TSG}$is the set of edges that cause a sequence operation termination.

The proposed methodology in this paper identities the set of all testing scenarios where $scn_i = scn_1, scn_2, scn_3 \dots \dots scn_m$ and the set of all nodes are also identified. Initially TSGcontains only the *StartState*; then each node of all $scn_i \in scn$ should be followed by its corresponding *NextState*, and the duplicates, if any, are removed. The *StartState* for different scenarios as illustrated in [1] is StateA and five *FinalStates* are B, C, D, E and F. An operation starts with an *InitialState* and undergoes a number of intermediate states due to the occurrence of various edges. Initial edge(s) can be easily detected from ERT. The algorithm of the generating *TSG* from a sequence diagram uses the ERT as input to get the sequence diagram testing scenario graph as the output. This process goes through the following steps:

Find out all edges.

1. Detect the initial edge after *InitialState* and check if it was visited before.
2. Detect the pre/post conditions of initial edge.
3. If it was visited before but it affects more than one edge, it is revisited again.
4. If it was visited before and does not affect more than one edge, go to next edge.
5. If the edge leads to *FinalState*, consider the path from *InitialState* to *FinalState* as one scenario.
6. If two or more edges are related by parallel relationship, connect all parallel edges by one circle notation at the beginning of process and in the end as well.

If two edges are related by option relationship, connect option edges by one decision notation in the beginning of

process.

Now, the TSG represents all of the sequence diagram combined fragments and sequence messages. The initial edge is detected by identifying its preconditions and post conditions. If the precondition of an edge is not emerging from post condition of another edge, it can be said that the former is an initial edge. E1 in ERT is initial edge because it affects the other edges but it is not affected by any other edges except the loop edge. In contrast, the final edge is affected by others but it cannot affect others except the loop edges.

4.2 Testing Scenarios

Testing scenarios generation is the main step towards the generation of correct test cases. The proposed technique enumerates all possible paths from the start edge to the final edge of the TSG to generate test scenarios which cover all edges. Each path then is visited to determine which Combined Fragment represents it. Figure 2 represents 25 messages $m_j(j = 1,2, \ldots 25)$ between two objects with guard conditions from which 25 corresponding edges $E_i(i = 1,2, \ldots 25)$ are obtained. The proposed methodology in this paper generates 19 testing scenarios automatically to cover all paths and operations as shown in Table 1. The variety of testing scenarios is a main challenge in this phase. Five main novel features (loop, par, alt, opt and break) are available in sequence diagram as shown in Figure 2. This causes a large number of Combined Fragments. The number of Combined Fragments directly affects the number of scenarios. Furthermore, the increase in the number of guard conditions (constraints) causes an increase in the number of testing scenarios. Each testing scenario starts with *StartState* and ends with *FinalState*. Each *FinalState* may be the end of one or more scenario. For example, *StateB* (forscn_1), StateC (forscn_2), StateD (forscn_3), StateE (for scn_4andscn_5) and StateF (for scn_6, , andscn_7, scn_8, scn_9) are FinalStates.

5. Test Cases Generation Approach

Test cases are used to detect faults in a software system. A software system is a combination of behavioural and structural information. Sequence diagram provides behavioural information. OCL is used to determine the pre/post conditions of the edges but even then, some other additional information is missing in a sequence diagram especially in the description of method constraint and type information. This additional information may be derived from the class diagram and then can be appended to each message. Therefore, the structure of each variable v in the method (e.g., name, type, value, constraint) is obtained. Here, type and constraint refer to the data type and the attribute constraints associated to a variable v:name. The type information is used to map each variable v:name to a range of values (min, max). This makes sure that each variable on a path from the initial edge to the final edge is mapped to a range of values. Furthermore, the constraint information is used to appropriately set the boundary values min and max. For example, the type information such as int to a variable X sets$(min_x, max_x) = (min_{Int}, max_{Int})$.

Where min_{Int} and max_{Int} are the boundary values. Let us consider$x \geq 5$ as the constraint associated to a variable x. In that case, (min_x, max_x) will be set to $5, max_{Int})$ which will be recognized as the initial domain. Figure 1 represents the class diagram determination to enrich the TSG by additional information.

Now, each testing scenario must be represented by one test case. For example, $scn_1$is presented by Test Case 1 in Figure 8. In this case, the variable coin has to be checked if an input is a coin or not. Actually, in this case, no variable type can be checked to ensure that. In other case studies, such as student registration system, the type of variable plays an important role to test the model. For example, for a message enterStudentAge (age: integer), if the user used decimal number to represent age, the test case must detect that error. If the testing scenarios in Table 1 are mapped directly to test cases, then we get 19 test cases to cover all model specifications but these many cases will not be sufficient to cover all sequence diagram features or all Combined Fragments behaviour. Recall that a test scenario is a sequence of operations starting with first edge and ending into final one, whereas a test case must examine the Combined Fragment and test whether it works well or not. The following sub-section presents the combination of testing scenarios to generate legitimate test cases.

5.1 Combining Testing Scenarios

The combination process is based on the similarity of functionality of scenarios. For example, the test scenarios 4, 5, 6 and 7 are similar in behavioural point of view but after E10, the edges 11, 12, 13 and 14 are fired in parallel. The work in [1] shows that each pair of edges (11, 12 AND 14, 15) are fired separately and are connected with selection node. To avoid confusion in how theses edges are related, SP node and EP node are used in this work. Subsequently, the combination of test scenarios is specified for parallel operations. The proposed algorithm, in this paper, identifies the parallel edges in each scenario, and then combines all of the scenarios that are same but different in parallel edges. Thus, the scenarios 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,

16, 17, 18 and 19 can be combined as shown in Table 2. Therefore, the numbers of test cases are reduced to 11 test cases instead of 19. This reduction is important and necessary to produce legitimate test cases because parallel operations can be covered only after combining the relevant testing scenarios. In contrast, the selection operations (alt, opt and break) and the loop testing scenarios need not be combined because these scenarios depend on the selection condition and, thus, these scenarios are independent

Table 1. Coffee machine testing scenarios

Scn ID	Testing Scenarios
Scn_1	$E_1 \rightarrow E_2 \rightarrow E_3$
Scn_2	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_5$
Scn_3	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8$
Scn_4	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
Scn_5	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{12} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
Scn_6	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{14} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
Scn_7	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{15} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
Scn_8	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{22}$
Scn_9	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{12} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{22}$
Scn_{10}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{14} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{22}$
Scn_{11}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{15} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{22}$
Scn_{12}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
Scn_{13}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$
Scn_{14}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{11} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
Scn_{15}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{12} \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$
Scn_{16}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{14} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
Scn_{17}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{14} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$
Scn_{18}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{15} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
Scn_{19}	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow E_{15} \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19} \rightarrow E_{20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$

5.2 Legitimate Test Cases

After combining testing scenarios, the focus is on generating legitimate test cases. The last step of test case generation is to convert the testing scenarios (after combinations) to actual test cases. The test scenarios which are already populated with the necessary information from two important design artefacts namely, sequence diagram and OCL. Third design artefact the class diagram has provided the information about types and constrains of parameters in a message. In sequence diagram, the method m represents the event from a sender class C1 to a receiver class C2. The method *m* in class C2 has signature which is defined in the class diagram. The method signature represents the name of the method, types of the parameter and the return type while the class attributes represent information about the instance variables such as their names and types. In addition, there may also be OCL constraints which are used to express invariants on the class attributes [8]. These invariants identify attribute constraints that are true for all instances of the class. Information Table (InfT) represents the relationships between the methods in sequence diagram and its signature in class diagram. This technique completes the full image of test cases. For each edge *Ej* of scenario scn_i the edge method *m* is identified. For the purpose of discussion, two cases are used in this paper: a method having a guard condition (see Notation 3) and another method without guard condition. If the method *m* has no guard condition, the method is represented in test case as follows:

$$TC = [preC, I(a_1, a_2 \dots \dots a_n), O(d_1, d_2 \dots \dots d_m) posC] \quad (3)$$

where, *TC*is a Test Case.

preC = precondition of the method m.

$I(a_1, a_2 \dots \dots a_n)$= set of input values for the method *n*.

$O(d_1, d_2 \dots \dots d_m)$= set of resultant values in the object when the method is executed.

posC= the post condition of the method *m*.

In contrast, when the method has a guard condition the test case uses this condition to show the state of processing. For example, the loop Combined Fragment in Figure 2 is affected by a guard condition that forces

the user to enter valid coins and the *coinAmount* is set to 50. This operation repeats until the suitable condition is reached that is acceptable according to the given guard condition and then it goes to the next step.

coinAmount<50 will be appearing in test cases until the event satisfied the guard condition. The test case with guard condition semantic in Notation 4 is similar in Notation 3, but the guard conditions *(g(v))* appeared in test case.

$$TC = \left[preC, I(a_1, a_2, ..., a_n), O(d_1, d_2, ..., d_m), g(v), posC\right]$$

Figure 4 shows the first 4 test case generated automatically to test the coffee machine model. We need to analyse whether all these test cases are legitimate.

Legitimate set of test cases refers to the ability of test case to achieve the best testing in minimum time and cost. Removing redundant test cases is one of the major challenges to obtain legitimate set of test cases. Redundant test case is one, which if removed, will not affect effectiveness of fault detection of the suite of remaining test cases [9]. In this paper, redundant test cases cannot be present because redundancy is detected automatically in the stage of TSG construction before generating testing scenarios. On other hand, if test cases are generated from only edges and nodes from sequence diagrams, the resultant suite may contain redundancy. For example, TC (4,5) and TC(6,7) contain similar edges except [E_{11}||E_{12}] and [E_{14}||E_{15}]. So, the difference between two Combined Testing Scenarios (CTS) occurs in the branches that are affected by *alt*. Furthermore, before and after the branches, the similarity between the CTSs can be seen. Unfortunately, test cases based on the scenarios are not able to eliminate redundant edges or nodes because each scenario must be tested separately. To minimise this problem, we propose an approach to select best testing scenario in the next sub-section.

5.3 Best Testing Scenario Selection Approach Using Genetic Algorithm

Test cases are derived from testing scenarios. An important question is what is meant by the best testing scenario in the sequence diagram? Coverage criteria play a major in selection of the best scenario. In this section, the coverage criteria that are used in this paper are listed. These criteria are used to evaluate the test cases that have been generated previously (see Figure 4). A total of 11 test cases satisfied our criteria. We propose the following coverage criterion which is expressed by the following condition.

Loop adequacy criterion: Test cases must contain at least one scenario test case in which control reaches the loop and then check the non-executable loop (zero iteration). Another test case for testing the body of the loop that is executed at least once before control leaves the loop (more than zero iteration) [10].

Concurrent (parallel) coverage criterion: For each parallel node in TSG, test cases must include one test case corresponding to every valid interleaving of message sequences.

All messages coverage criterion: For all messages in sequence diagram, test cases must execute all message sequence paths of the sequence diagram [11].

Branch coverage: Each decision outcome within the diagram must be covered by at least one start-to-end path [26].

Selection coverage criterion: For each selection fragment (alt, opt and break), test cases must include one test case corresponding to each evaluation of the constraint.

The criteria above are covered by test cases that are generated in this paper, but what is the best test case, which covers all criteria?. When we say "best test case" we mean the one which has the highest percentage of coverage. To select the best test case, the criteria mentioned above are categorised into two evaluation steps: first one includes all of the above coverage criteria except the “All message coverage criterion” and we call it Combine Fragments Coverage Criterion (CFCC), and the second one is specified for “All messages coverage criterion”. Figure 5 shows the uses of these two criteria to select the best CTS.

Table 2. Combined testing sentences

CTSj	Testing Scenario
CTS_1	$E_1 \rightarrow E_2 \rightarrow E_3$
CTS_2	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_5$
CTS_3	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8$
CTS_4 (4,5)	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{11} \| E_{12}] \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
CTS_5 (6,7)	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{14} \| E_{15}] \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}$
CTS_6 (7,8)	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{11} \| E_{12}] \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}\ E_{20} \rightarrow E_{21} \rightarrow E_{22}$

$CTS_7(10,11)$	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{14} \parallel E_{15}] \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19}\ E_{20} \rightarrow E_{21} \rightarrow E_{22}$
$CTS_8(12,14)$	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{11} \parallel E_{12}] \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19\ 20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
CTS_9 (13,15)	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{11} \parallel E_{12}] \rightarrow E_{13} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19\ 20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$
CTS10 (16,18)	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{14} \parallel E_{15}] \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19\ 20} \rightarrow E_{21} \rightarrow E_{23} \rightarrow E_{25}$
$CTS_{11}(17,19)$	$E_1 \rightarrow E_2 \rightarrow E_4 \rightarrow E_6 \rightarrow E_7 \rightarrow E_8 \rightarrow E_9 \rightarrow E_{10} \rightarrow [E_{14} \parallel E_{15}] \rightarrow E_{16} \rightarrow E_{17} \rightarrow E_{18} \rightarrow E_{19\ 20} \rightarrow E_{21} \rightarrow E_{24} \rightarrow E_{25}$

To achieve this target, the proposed approach using GA aims to generate testing scenarios to cover maximum Edges using genetic algorithms technique. The “All message coverage criterion” depends on the weights of the edges. The weight of the edges is calculated by the following Equation:

$$E_i w = \frac{E_p}{N} \tag{1}$$

$E_i w$is the weight of each edge.

E_p is the edge presence probability in whole testing scenarios.

N is the total weights of all edges in all testing scenarios.

For example, the repeated use of the E10 is 8 times over all combined testing scenarios in Table 2. The weight of E10 is calculated by Equation 1 as follows,

$$E_{10} w = \frac{8}{147} = 0.54421769$$

<table>
<tr><td>Precondition</td><td colspan="2">Coffee Machine displaying main screen</td></tr>
<tr><td rowspan="3">Test Case 1</td><td>Input:</td><td>coin: inserCoin (): isCoin (coin) ="False"</td></tr>
<tr><td>Output:</td><td>coinBack (coin)</td></tr>
<tr><td>Postcondition:</td><td>Display main screen</td></tr>
<tr><td rowspan="3">Test Case 2</td><td>Input:</td><td>coin: coinAmount: : inserCoin (): isCoin (coin) ="True": isEnoogh (coinAmount) ="false"</td></tr>
<tr><td>Output:</td><td>moreCoin (coin)</td></tr>
<tr><td>Postcondition:</td><td>Display more coin message: “coinAmount<50”</td></tr>
<tr><td rowspan="3">Test Case 3</td><td>Input:</td><td>coin: coinAmount: water: : inserCoin (): isCoin (coin) ="True" isEnough (coinAmount) ="True": ok (): fillWater (water)</td></tr>
<tr><td>Output:</td><td>Wait ()</td></tr>
<tr><td>Postcondition:</td><td>Display wait message</td></tr>
<tr><td rowspan="3">Test Case 4</td><td>Input:</td><td>coin: coinAmount: water: coffee: HotCh: inserCoin (): isCoin (coin) ="True": isEnough (coinAmount) ="True": ok (): fillWater (water): getWater (water) ="true": Order (): Ordr.coffee () ="True" [fillCoffee (coffee) || warmUp ()]: coffeeOk (coffee) ="True": tempOk () ="True": stratPour (coffee)</td></tr>
<tr><td>Output:</td><td>Pour (coffee)</td></tr>
<tr><td>Postcondition:</td><td>Display wait message</td></tr>
</table>

Figure 4. First 4 test cases for Coffee Machine system

GA is iterative procedures which work with chromosomes. The chromosomes are a population of candidate solutions that are maintained by the GAs throughout the solution process [9]. At first a population of chromosomes is generated randomly. A selection operator is used to choose two solutions from the current population. The selection process used the measured goodness of the solutions (Fitness Function). The crossover operator swaps sections between these two selected solutions with a defined crossover probability. One of the chromosomes solutions is then chosen for application of the mutation process. The algorithm is ended, when a defined stopping criterion is reached. Since our approach focuses on finding a set of transitions by triggers firing, a chromosome in our approach is a sequence of triggers itself. Testing scenarios will be the first population from all possible cases. The fitness value for each chromosome is calculated from the number of Edges which is covered.

GAs operators used in proposed technique are the two point crossover and random mutation. Based on some experimentation and previous knowledge on GAs application with other problem [20], the parameters of genetic operation are set as follow:

- The crossover probability is 0.5.
- The mutation probability is 0.05.
- The size of population in each generation is 10.

The All messages coverage criterion uses the weight of the edges to calculate the weight of a CTS. This weight is used to identify the best scenario. The intuition is that such a CTS makes best messages coverage. Equation 2 shows that the weight of a scenario is the sum of weights of all edges in that scenario. The proposed methodology shows the combined testing scenarios CTS8, CTS9, CTS10 and CTS11 have the highest weight which means that these four CTSs are the best from message coverage point of view.

$$CTS_i w = \sum_{i=0}^{n} Ew_{i+1} \quad (2)$$

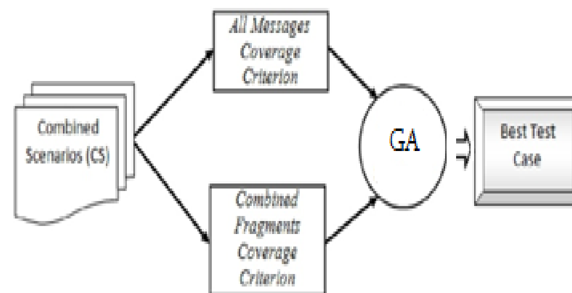

Figure 5. Best test case selection methodology

Note that we have not adopted the *All messages coverage criterion* as the only factor to select the best test scenario. Second factor (CFCC) is also used to check the Combined Fragments coverage. Based on this factor, the best CTS is that one which is able to cover the highest number of Combined Fragment's edges.

By referring to Figure 2, it can be noted that the *Combined Fragments* (*loop*, *alt*, *par*, *opt* and *break*) contain 13 edges (E_1, E_2, E_3, E_4, E_5, E_{11}, E_{12}, E_{13}, E_{14}, E_{15}, E_{16}, E_{22} and E_{24}). The best CTS is the one that covers highest number of these edges. Thus, the proposed approach in this paper examines the presence of Combined Fragment edges in each testing scenario. CTS_{10} covers 6 Combined Fragments edges (E_1, E_2, E_4, E_{14}, E_{15} and E_{16}) out of 13. The CTSs with the best Combined Fragments coverage are CTS_6, CTS_7, CTS_9 and CTS_{11}.

The best combined testing scenario (s) is that one which achieves highest coverage in both the criteria *CFCC* and *All messages coverage criterion*. CTS_9 or CTS_{11} can be selected as best *CTS* because these two have the highest coverage in both these criteria. Let us say CTS_{11} is the best test scenario and the corresponding test case covers 88% of *All messages coverage criterion* and 55% of *CFCC*. As noted, the *CTS*is unable to capture the coverage of enough edges in the Combined Fragment as just 55% coverage may not be satisfactory.

To improve the coverage of the Combined Fragments edges, we propose to select the second best combined testing scenario as well to support the first one. This technique is similar to that used in [1] to select second testing path to improve unit coverage. The selection method for the second best *CFCC* scenario depends on the non-similarity of edges contained in the best *CFCC* scenarios. In [1], the non-similarity criterion is used to select the second best testing path, and the same is used in this paper as well. Based on the non-similarity degree between the best *CFCC* scenario and others, the scenarios which are eliminated have the biggest similarity degree. The CTS_6 is the highest non-similar scenario in comparison with CTS_{11} because it has 4 different Combined Fragment's edges (E_{11}, E_{12}, E_{13} and E_{22}) compared to CTS_{11}. Consequently, both scenarios CTS_{11} and CTS_6 cover 85% of Combined Fragments edges and 97% of *All messages coverage criterion*. Together they are considered as best testing scenarios.

6. Conclusion and Future Work

An approach is proposed to generate test cases automatically from UML sequence diagram, class diagram and OCL. The approach generates efficient test cases that meet required coverage criteria. Furthermore, the relationship between edges has been recognised by the edge relationships table. This table is generated automatically to reveal the relationship of the edges as specified in the sequence diagram. The proposed methodology covers *loop*, *parallel*, *alternative*, *option*, *break* and *sequence* relationships and a number of structural specifications as well. This work provides an efficient technique to generate testing scenarios graph

that is used to represent the testing scenarios of the events in a sequence diagram. The testing scenario graph represents combined fragments and shows the relationships among scenarios. The proposed methodology decomposes testing scenarios graph to combined test scenarios to achieve minimum number of testing scenarios. Information Table (**InfT**) technique is proposed to provide the sequence diagram with exact method signatures that are obtained from the class diagram. Despite this, presence of the redundant edges in each testing scenario led us to develop a new technique to select best testing scenario. To achieve this, we have used *All messages coverage criterion* and *Combined Fragments coverage criterion* to evaluate the best testing scenario (and the best test case) using GA. The weakness that appeared in selecting best testing scenario is a variety of the coverage percentage that the best testing scenario can cover. This variety comes from the variety of the model designs. Generally, this technique almost covers more than 85% of Combined Fragment's edges and more than 94% of the sequence diagram messages.

The test cases that are generated automatically in this paper meet the coverage criteria. However, UML sequence diagrams do not contain all information related to verification and non-functional parameters of the software. This limitation comes from the semi-formal characteristics of UML diagrams [12], especially during the first cycle of the software production. In order to avoid its semi-formal characteristics, we propose to transform sequence diagram and class diagram to HLPN, a type of Petri Net in future. Petri Net (PN) is a graphical diagram for the formal description of the flow of activities in complex systems [13].

References

Alhroob, A., Dahal, K., & Alamgir, H. (2009). *Automatic Test Cases Generation from Software Specifications Modules* (pp. 130-142). In Proceedings of the 4th IFIP TC2 Central and East European. Conference on Software Engineering Techniques CEE-SET. Springer.

Andrews, A., France, R., Ghosh, S., & Craig, G. (2003). Test adequacy criteria for UML design models. *Software Testing Verication and Reliability, 13*(2), 95-127. http://dx.doi.org/10.1002/stvr.270

Aredo, D. (2002). A framework for semantics of UML sequence diagrams in PVS. *Journal of Universal Computer Science, 8*(7), 674-697.

Bobbio. (1990). System modelling with Petri nets," in Systems reliability assessment: proceedings of the Ispra course held at the EscuelaTecnica Superior de IngenierosNavales, Madrid, Spain, September 19-23, 1988, in collaboration with Universidad Politecnica de Madrid. Springer, p. 103.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). UniedModeling Language User Guide, The (Addison-Wesley Object Technology Series).

Cavarra, A., & Kuster, F. J. (2005). Combining sequence diagrams and OCL for liveness. *Electronic Notes in Theoretical Computer Science, 115*, 19-38. http://dx.doi.org/10.1016/j.entcs.2004.09.025

Debasish, K., Debasis, S., & Rajib, M. (2013). Automatic code generation from unified modelling language sequence diagrams. *IET Software, 7*(1), 12-28. http://dx.doi.org/10.1049/iet-sen.2011.0080

Edvardsson, J. (1999). *A survey on automatic test data generation* (pp. 21-28). In Proceedings of the 2nd Conference on Computer Science and Engineering.

Federico, C., Antonio, C., & Toni, S. (2010). Automating Test Cases Generation: From xtUML System Models to QML Test Models. ACM proceeding in MOMPES ’10, September 20, 2010, Antwerp, Belgium.

Gordon, F., & Andreas, Z. (2012). Mutation-Driven Generation of Unit Tests and Oracles. IEEE *Transactions On Software Engineering, 38*(2), 287-292. March/April.

Koochakzadeh, N., & Garousi, V. (2010). A Tester-Assisted Methodology for Test Re-dundancy Detection. *Journal on Information and Software Technology, 52*(5), 625-640.

Li, B., Li, Z., Qing, L., & Chen, Y. (2007). *Test Case Automate Generation from UML Sequence Diagram and OCL Expression* (pp. 1048-1052). in Proceedings of the 2007 International Conference on Computational Intelligence and Security. IEEE Computer Society.

Li, X., Liu, Z., & Jifeng, H. (2004). A Formal Semantics of UML Sequence Diagram," in Proceedings of the 2004 Australian Software Engineering Conference. IEEE Computer Society, p. 168.

Mahesh, S., Amit, S., & Rajeev, K. (2011). *Generation of Improved Test Cases from UML State Diagram Using Genetic Algorithm* (pp. 125-134). Proceeding of the 4th India Software Engineering Conference.

Motameni, H., Movaghar, A., Daneshfar, I., & Zadeh, H. (2008). Mapping to Convert Activity Diagram in Fuzzy UML to Fuzzy Petri Net. *World Applied Sciences Journal, 3*(3), 514-521.

OMG. (2010). *Object constraintlanguage, version 2.2.* Retrieved August 10, 2010, from http://www.omg.org/spec/OCL/2.2.htm

PreetiGulia, R., & Chillar, S. (2012). A New Approach to Generate and Optimize Test Cases for UML State Diagram. *ACM SIGSOFT Software Engineering Notes archive, 37*(3), 1-5.

Shaukat, A., Lionel, C., Briand, H. H., & Rajwinder, K. P. W. (2010). A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions On Software Engineering, 36*(6), November/December.

TarkanTekcan, V. Z., VukotaPekovic, N. T., & Mustafa, G. (2012). User-driven Automatic Test-case Generation for DTV/STB Reliable Functional Verification. *IEEE Transactions on Consumer Electronics, 58*(2), 587-595. http://dx.doi.org/10.1109/TCE.2012.6227464

utt, A. O., Jin, Z., & Pan, J. (1999). The dynamic domain reduction procedure for test data generation. *Software: Practice and Experience, 29*(2), 167-193.

ValentinDallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, & Andreas Zeller, Member, (2012). Automatically Generating Test Cases for Specification Mining. *IEEE Transactions on Software Engineering, 38*(2). MARCH/APRIL.

Copyrights

Copyright for this article is retained by the author(s), with first publication rights granted to the journal.

This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).