A NOVEL METHODOLOGY FOR TEST SCENARIO GENERATION BASED ON CONTROL FLOW ANALYSIS OF UML 2.X SEQUENCE DIAGRAMS

Saroj Kanta Misra[1], Durga Prasad Mohapatra[2]

[1] *Assistant professor, Department of IT, GIET, Gunupur, Odisha, India*
[2] *Associate Professor, Department of CSE, National Institute of Technology, Rourkela, Odisha, India*

Abstract

Now a days UML is widely used for preparing design documents. It helps to specify, construct, visualize and document artifacts of software systems. This paper presents an approach to test the software in the early stage (design phase) of software development life cycle, so that it can help the software testers in the later stages. This paper focuses on generating test scenarios from UML 2.x Sequence diagrams. The most challenging problem in generating test scenarios from UML 2.x sequence diagram is the presence of fragments such as alt, loop, break, par, opt etc. We propose an intermediate control flow graph in a testable form named Sequence Control Flow Graph (SCFG) resulting from the control flow analysis of UML 2.x sequence diagrams. We also propose a systematic approach named Sequence Test Scenario Generation Algorithm (STSGA) for generating test scenarios from UML 2.x Sequence diagrams. The test scenarios generated by our approach are suitable for detection of scenario faults, use case dependency and system testing.

Key Words: *Test scenarios, UML 2.x Sequence diagram, Sequence Control Flow Graph (SCFG), Test Scenario Generation Algorithm (STSGA).*

--***--

1. INTRODUCTION

Nowadays, due to rapid increase in size and complexity of software applications, more emphasis is given towards object-oriented design strategy, which helps to reduce software cost and increase software reliability, and usability. But, introduction of object-oriented design and implementation approach brings out some new difficulties for software testing. Several features of object-oriented approach like polymorphism, dynamic binding, inheritance etc. create certain difficulties in software testing process. To test such object-oriented software from their implementation code is a very a complex process due to the different features of object oriented approach. Model Based Testing of these object-oriented software can be beneficial to detect the error in the design phase itself, so that these error do not propagate to other stages of software development life cycle. Control Flow Analysis (CFA) plays a vital role in determining all possible alternative paths a program may follow during execution. A Control Flow Graph (CFG) is a static representation of a program that represents all alternatives of control flow. For example, both choices for *If-else* statement can be represented in CFG as different control flow paths. A *Loop* can be represented as a cycle in a CFG.

According to Garousi et al. [1] Control flow information can be derived from two different sources: from software design artifacts and code itself. In Code-based CFA(CBCFA), control flow information is obtained from the available source code, whereas in Model-based CFA(MBCFA), control from information is obtained from design models such as UML. The motivation of our work is to derive control flow information and generate test cases in the early stage of software development life cycle, after the UML design models of a system become available.

2. BASIC CONCEPTS AND DEFINITIONS

UML behavioral diagrams such as interaction diagram, activity diagram, and state machine diagram describe different functionalities of the system and also capture various dynamic behavior of the system. Interaction diagrams illustrate the system functionalities using different fragments such as alternative, loop, break, parallel, etc. In this section we provide an overview of interaction diagram, XMI and Sequence Control Flow Graph (SCFG), how to obtain a SCFG.

2.1. UML 2.x Interaction Diagram

Interaction diagrams describe how a group of objects collaborate in some behavior - typically a single use-case. Interaction diagrams are of two types: Sequence diagram and Communication diagram. The basic objective of both the diagrams is same. Sequence diagrams accentuate on the time sequence of messages passed between the communicating objects, and the communication diagrams accentuate on the structural organization of the communicating objects that send and receive messages. In our approach, we have used UML 2.x Sequence diagrams to generate test scenarios.

2.1.1 New Features of UML 2.x sequence diagram

List below describes some of the feature of UML 2.O sequence diagrams

- **Interaction:**

Series of messages that is passed between different communicating objects to satisfy some task is called interaction.

- **Interaction Occurrences:**

When an interaction used within another interaction or context, then it is called interaction occurrence.

- **Combined Fragments:**

Combined fragment is an interaction fragment which is a combination of multiple interaction fragments. Each combined fragment has an interaction operator and corresponding interaction operands.

- **Interaction Operand:**

Interaction operand shows grouping of interactions within the combined fragment.

- **Interaction Operator:**

Interaction operator for combined fragments describes how the interaction operands present inside the combined fragments are going to be used. The followings are the list of interaction operators defined.

- **Alternative (alt):**

The interaction operator alt works like if-then-else structure. At most one operand can be selected based on the guard expression's true value. If there is no guard expression, then an implicit true guard value is implied.

- **Option (opt):**

The interaction operator opt is used, when the combined fragment represents the operand as an option where the operand either happens or may not happen. It works like an alternative combined fragment where one operand is nonempty and the other one is empty.

- **Loop (loop):**

The interaction operator loop represents a loop structure. The interaction operand present inside the loop combined fragment will be repeated many times. The repetition of loop can be controlled either or both by iteration bound and guard. If a loop combined fragment has no bound specified, then the loop will execute with infinite as upper bound and zero as lower bound.

- **Break (break):** The interaction operator break is used to represent a breaking or exceptional scenario to be performed instead of the remaining interaction fragment.

- **Parallel (par):** The interaction operator par describes parallel execution of behaviors of the interaction operands present inside a combined fragment.

2.2 XMI

XMI (XML Metadata Interchange), is an extension of XML that facilitates the standardized way for interchanging object models and metadata. Specifically, XMI is useful to programmers using the Unified Modeling Language (UML) with various languages and development tools to exchange their data models with each other.

2.3 Sequence Control Flow Graph (SCFG)

In order to examine and visualize the control flow information present in the UML sequence diagram, we first extract all the control flow information from the XMI equivalent of the sequence diagram, then we construct an intermediate control flow graph in a testable form called Sequence Control Flow Graph (SCFG). As UML sequence diagram contains information about objects of a system in form of messages in a time sequence and we focus on functional testing of the system, we will not give emphasis on messages sent between internal objects. For each message that is sent from internal objects to user object, our goal is to generate test scenarios taking these messages as end points. So, we will obtain SCFG, where nodes represent messages in sequence diagram and edges represent path between nodes. The messages sent from internal objects to user object are colored gray. We have added some additional nodes for the sake of simplicity in the process of generating test scenarios.

The following are the type of nodes considered for constructing Sequence Control Flow Graph (SCFG).

Definition: An Sequence Control Flow Graph is a tuple $R = \{R, M, F_{start}, F_{end}, E_{output}, C, E\}$ where,

- R is the root node of the Sequence Control Flow Graph (SCFG).
- M is a message node that represents a message from UML sequence diagram.
- F_{start} (*Fragment start*) is a set of nodes representing the starting of a fragment.
- F_{end} (*Fragment end*) is a set of nodes representing the End of a fragment.
- E_{output} (*Expected output)* is the set of nodes that precedes the message from internal object to user object in Sequence Control Flow Graph(SCFG).
- C (*Condition node*) is the set of nodes representing conditions for the fragments.
- E is the set of final nodes representing an exit of Sequence Control Flow Graph (SCFG).

2.4 Constructing the Sequence Control Flow Graph (SCFG)

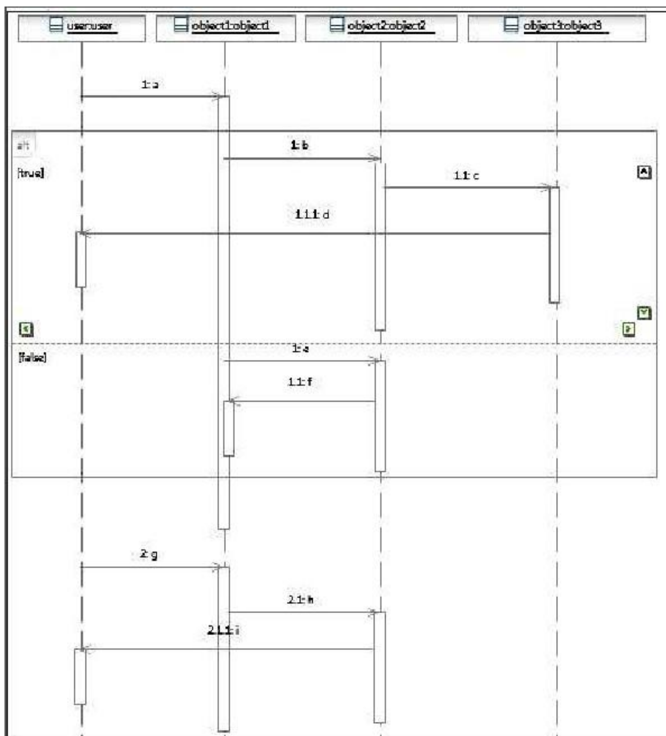

Fig.1. A sample Sequence diagram

We construct the SCFG for representing control flow among messages in presence of fragments and nested fragments. In this process, we give more emphasis on use scenarios [3] (actions executed by the user and actions viewed by the user).

Each message present in the sequence diagram is represented by a node in SCFG. The start and the end of every fragment is denoted by two additional *fragment* nodes representing starting and ending of fragment such as *alt_start, alt_end, par_start, par_end etc*. In alt fragment, the conditions for control flow are also denoted by additional $control$ nodes containing a condition sequence number and condition itself such as condition1_true, condition2_false, etc. The steps to build a SCFG from a sequence diagram are presented as fallows.

- The root node of SCFG is represented by a node *start.*
- The end points of SCFG are represented by the node *end.*
- From the root node *start*, for each message in the sequence diagram, a new node is added into the SCFG with it's value same as message name in the sequence diagram.

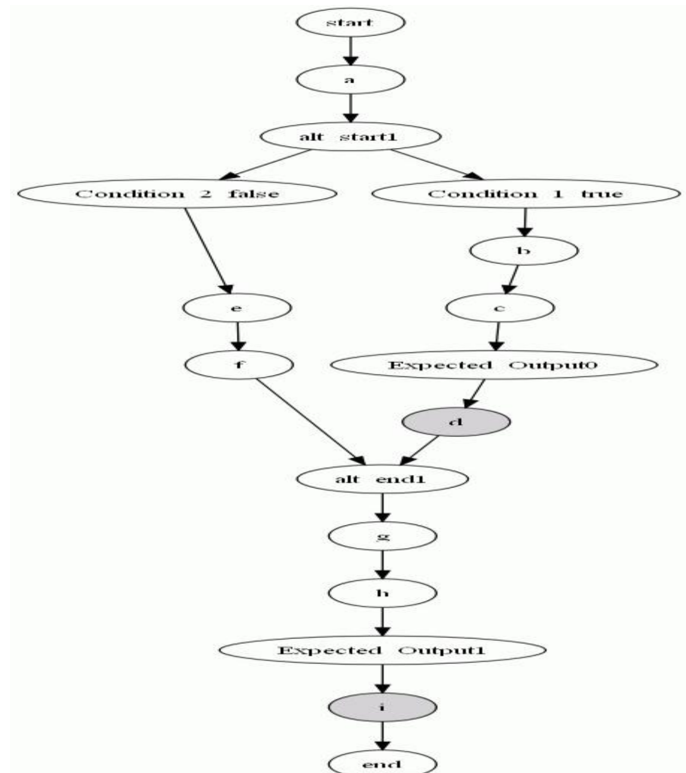

Fig.2. Derived SCFG from UML sequence diagram given in Figure 1

- For each message (in order it appearing sequence diagram) do the following :
  1. If the message is from user object to non-user object (internal object) or from non-user object to non-user object then a new node is added into SCFG with a directed edge from its previous node to itself.
  2. If the message is from non-user object to user object then two nodes are added into SCFG (1) first node with value *expected_output* and a directed edge from its previos node to itself. (2) second a gray color node with value same as message name and a directed edge from *expected_output* node to itself.

Figure 1 shows an example UML 2.x sequence diagram where message passing occurs between user object and various internal objects of the system. The corresponding SCFG for Figure 1 is given in figure 2. We can observe from the SCFG in Figure 2, that all the messages of sequence diagram are represented as nodes and the starting of *alt* fragment is represented using a *Fragment start* node alt start1 and ending using a *Fragment end* node alt end1. Both the conditions for alt fragment are represented by *Condition* nodes Condition 1 True, Condition 2 False. The gray colored nodes represent messages that are passed from the internal objects to the user objects.

3. PROPOSED METHODOLOGY

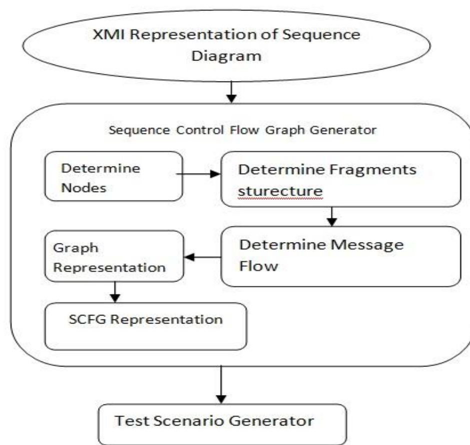

Fig. 3. Block diagram for generating test scenarios from UML 2.x sequence diagram

In this section we discuss our approach for test scenario generation from UML 2.x sequence diagrams. Our sequence diagram includes combined Fragments using various Intraction Operators such as *alt, par, loop, break* etc. We propose a mechanism to extract the messages in their timing sequence and Fragments precisely from XMI representation of UML 2.x sequence diagram. Then we map every message to its corresponding fragment. Next we generate the Sequence Control Flow Graph (SCFG) using Sequence Control Flow Generator. Then, we generate test scenarios using Test Scenario Generator .The block diagram for generating test scenarios from sequence diagram is given in Figure 3.

The major steps of our approach are partitioned into three phases. They are given below:

- Parsing the XMI representation of UML 2.x Sequence Diagram.
- Developing the Sequence Control Flow Graph (SCFG) generator.
- Developing the Test case generator.

The first step of our approach is to parse the XMI representation of UML 2.x sequence diagram. We have used IBM Rational Rose Architecture (RSA) to draw the sequence diagram, and then we have exported the XMI representation for the sequence diagram, which is used as the input for our procedure. We propose a parser that parses the XMI file to extract information about messages, structure of fragments and combined fragments. Messages that are sent from any non user object to user object are identified. Using all the extracted information, a Sequence Control Flow Graph is generated.

In order to generate test scenarios for the sequence diagram, the SCFG needs to be traversed. We propose a Sequence Test Scenario Generation Algorithm (STSGA) for generating test scenarios. The detailed algorithm is shown in Algorithm 1. This algorithm traverses the Sequence Control Flow Graph (SCFG) to first identify the the messages that are .rom non user object to user object. Then it generates the test paths from start node to the nodes connected to these messages and then finally generates test scenarios for these messages.

4 IMPLEMENTATION OF OUR ALGORITHM

In this section, we exemplify our approach for generating test scenarios form XMI representation of UML sequence diagram by converting XMI representation of UML sequence diagram into an equivalent Sequence Control Flow Graph (SCFG) and then generating test scenarios. We generate test scenarios from UML sequence diagram to test the feasibility and concurrency errors.

We have developed a prototype tool called XMI2SCFG (XMI to Sequence Control Flow Graph) for generating test scenarios. XMI2SCFG works in two steps (1) Parsing of XMI representation of UML Sequence diagram.(2) Creating a SCFG (Sequence Control Flow Graph) in image format, and generating test scenarios from SCFG. We have implemented XMI2SCFG in Java language using Netbeans IDE 7.0.1. XMI\-2\-SCFG takes the XMI representation of UML 2.x sequence diagram as input. We have used IBM Rational Software Architecture (RSA) 7.0 to draw the sequence diagram and then exported the XMI representation (XMI equivalent of UML sequence diagram).

In the first phase, XMI2SCFG uses SAX parser to parse the XMI representation of sequence diagram. Along with the main class and sequenceParser some auxilary classes such as *listMsg, listAlt, listPar, listLoop, listBreak* are used for this propose. The sequenceParser class implements various methods such as *getMsgID(), getMsgName(), getUserMsg(), getAlt-Msg(), getBreakMsg(), getPar-Msg(), getLoop-Msg()* to interact with SAX parser. These methods process the tagged elements present in XMI representation of UML sequence diagram such as *"packagedElement", "message", "fragment", "ownedAttribute","`lifeline", "operand", "guard", "body", ``ownedOperation"* to extract various information like message name, message flow, message dependacies, message type, guard condition, etc.

In the second phase,the task of XMI2SCFG is to visualize the SCFG (Sequence Control Flow Graph) in an image format. For SCFG visualization two main classes are used : DotTransformer and Graphvizvisualization. We have used for Graphviz. Taking two linked lists tranSource and tranDestination as input, the DotTransFormer objet creates a .dot file . After the .dot file is created, the methods present in graphiviz getDotSource(), getGraph(), and writeGraphTofile() create an image for SCFG (Sequence Control Flow Graph). Then the SCFG is supplied as input to Sequence Test Scenario Generation (STSG) Algorithm which generates the test scenarios. Starting from the root node $start$, STSGA scans each node of the SCFG, depending on the node type such as *message node, Fragment node, Condition node,* etc each node is processed differently in STSGA. Finally STSGA generates a set of test scenarios for UML sequence diagram.

Algorithm 1 Sequence Test Scenario Generation Algorithm (STSGA)

```
Input: Sequence Control Flow Graph (SCFG).
Output: Set of test scenario.

1: runningStack=ϕ
2: decisionStack=ϕ
3: userMessageStack=ϕ
4: resultStack=SCFG.rootNode
5: for all nodes of SCFG do
6:     while SCFG.node==expextedOutput.node do
7:         userMessageStack.push(child node of SCFG.node)
8:     end while
9: end for
10: for all elements of userMessageStack do
11:     repeat
12:         if runningStack[top] ≠ alt.node || loop.node|| par.node|| break.node || SCFG.EndNode then
13:             resultStack.push(runningStack.pop) ▷ push runningStack top element in resultStack and pop the top element from runningStack.
14:             resultStack.push(child node of resultStack[top] in SCFG)    ▷ push child node of resultStack top element into resultStack.
15:         else if runningStack.[top] == SCFG.EndNode then
16:             Mark the last decision node in resultStack as visitd
17:             while resultStack[top] ≠ decisionStack[top] do
18:                 resultStack.pop    ▷ pop the top element fom resultStack.
19:             end while
20:             decisionStack.pop    ▷ Pop the top element from decisionStack.
21:         else if runningStack[top] == alt.node || break.node || loop.node then
22:             decisionStack.push(runningStack.pop)    ▷ pop top element of runningStack and push it into decisionStack and resultStack.
23:
24:             for all Child nodes of resultStack[top] in SCFG do
25:                 if Child node is not Marked Visited then runningStack.push(Child Node) ▷ push all child nodes of resultstack[top] in SCFG,
    if marked as visited insert into runningStack.
26:                 end if
27:             end for
28:         else if runningStack[top] == par.node then
29:             resultStack.push(resultStack[top] and all its child nodes in SCFG)
30:             runningStack.push(child nodes of resultStack[top] in SCFG)
31:         else if runningStack.top == UserMessageStack.CurrentNode then
32:             resultStack.push(runningStack.pop)    ▷ Pop the top element from runningStack and push it into resultStack.
33:             Print resultStack
34:             if decisionStack ≠ ϕ then
35:                 while resultStack[top] ≠ decisionStack[top] do
36:                     resultStack.pop    ▷ pop the top element fom resultStack.
37:                 end while
38:             end if
39:             if decisionStack ≠ ϕ then decisionStack.pop    ▷ pop the top element from decisionStack.
40:             end if
41:         end if
42:     until runntingStack ≠ ϕ & decisionStack ≠ ϕ
43:     delete all elements from resultStack
44:     delete all elements from decisionStack
45:     delete all elements from runningStack
46: end for
47: Exit
```

5 CASE STUDY

In this section, we illustrate the working of our approach for generating test scenarios with the help of a case study pertaining to Restaurant Automation System (RAS). The RAS automates different functionalities of a Restaurant such as take order, order processing, Generate Bill etc. We are considering a particular use case, "Generate Bill ". In the use case Generate Bill, The manager inputs the order number to generate the bil. Based on the current status of the order, different scenarios are possible such as bill already generated, order is not processed etc.

The sequence diagram of Generate Bill usecase is given in Figure 4. The sequence diagram for Generate Bill use case contains two alt (alternate) fragment and two par (parallel) fragment, two loop fragment and two break fragment. . In the Generate Bill use case sequence diagram the messages DisplayMessagee(Bill will be generated after delivary), DisplayMessage(Bill_is already generated), DisplayMesage(Bill Number, Bill Amount) and DisplayMessages(Oreder is not found while generating the bill) are the messages the user receives from internal objects. The complete Sequence Control Flow Graph (SCFG) generated for the Generate Bill use case of the sequence diagram in Figure 4 is given in Figure 6. Each message that the user object receives from internal objects is represented by gray colored nodes preceded by expected output nodes.

Table 1 shows the test scenarios are which obtained by supplying SCFG as input to our STSG

6 COMPARISONS WITH RELATED WORK

In this section, we discuss some existing approaches similar to our approach for test scenario generation from sequence diagrams. In many cases test scenario generation is done manually, as in case of many related work. For large systems, it is practically impossible to generate test scenarios manually from sequence diagrams.

Sarma et al. [2] proposed an approach to generate test scenarios from UML sequence diagram, by converting sequence diagram into an directed graph called Sequence Diagram Graph (SDG), where a nodes in SDG represents a message in the sequence diagram and a directed edge represent control flow between the nodes. SDG is then used to generate test scenarios. Sarma et al.\cite{Ref2} used UML 1.x sequence diagram for their work, which did not support fragments such as alt, par, loop, break etc whereas our approach considers these fragments by using UML 2.x sequence diagram.

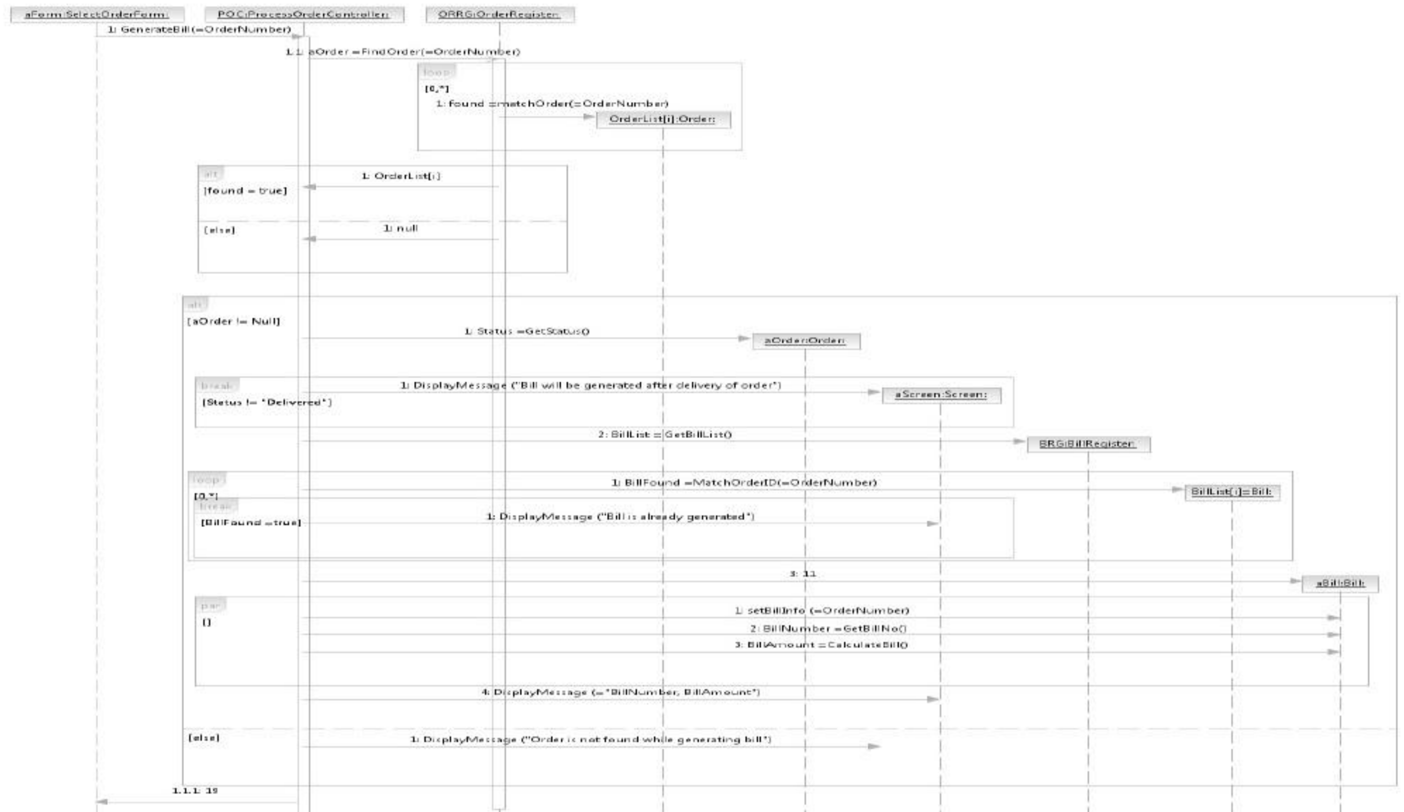

Fig.4. Sequence Diagram of Generate Bill use case

Fig.5. A portion of the SCFG for Generate Bill uses case and test scenarios using XMI2SCFG

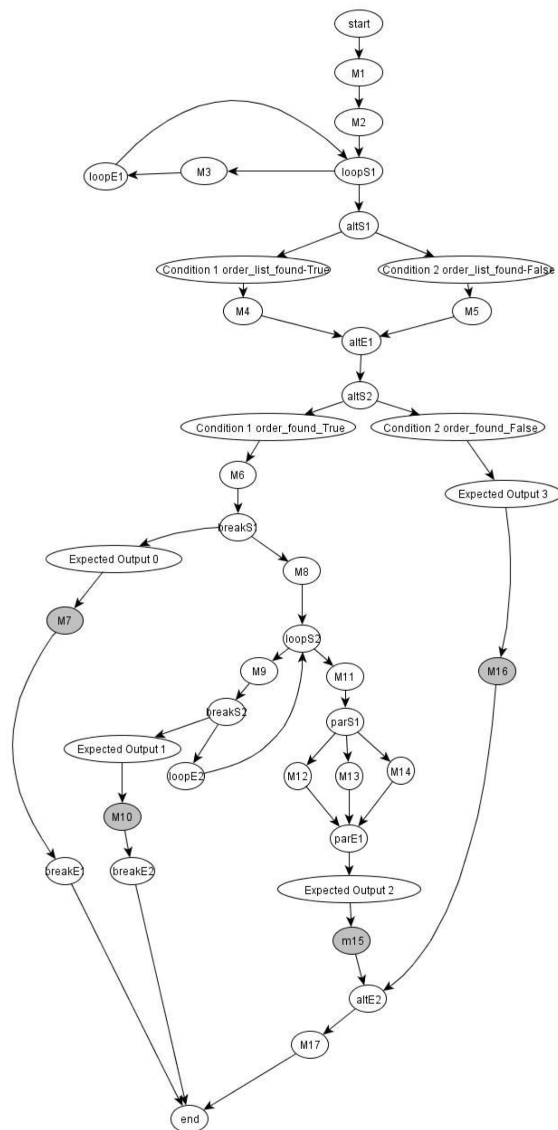

Fig.6. The complete SCFG of Generate Bill use case of the sequence diagram given in figure 4

Cartaxo et al. [3] proposed a approach to generate test paths for mobile application using sequence diagram. They, constructed an intermediate model call Labeled Transition System (LTS) from sequence diagram, where directed edges ware used to represent control flow, expected output. Then, they have applied depth first search (DFS) algorithm to traverse the LTS model for generating test paths. However, their approach did not support fragments present in UML 2.x sequence diagram.

Khandai et al. [4] proposed another approach to generate test cases from sequence diagrams. They had constructed an intermediate graph called Concurrent Composite Graph (CCG) generated from sequence diagram, which was a variant of activity diagram. Then they traversed the CCG by applying depth first search (DFS) and breath first search (BFS) to generate test cases. They have used BFS algorithm to explore *fork* and *joint* constructs.

We proposed a novel approach to test object-oriented software based on control flow analysis of UML sequence diagrams. Our approach is a fully systematic approach for automatic test scenario generation from UML 2.x sequence diagram, which supports various fragment such as *alt, par,* etc.

7. CONCLUSIONS

In this paper, we have proposed a novel approach for test scenario generation from UML 2.x sequence diagram considering the fragments, nesting of fragments and control flow primitives present in sequence diagrams. The method first generates an intermediate graph called Sequence Control Flow Graph (SCFG) from the XMI representation of UML 2.x sequence diagram. Then by analyzing the control flow information, message sequence and the fragment structure, our proposed approach generates test scenarios, for various use case present in a system. Most of the existing techniques of test scenario generation from UML sequence diagrams are manual and do not consider fragments and nesting of fragments into test scenarios. Hence, these methods become more complex while taking UML 2.x sequence diagrams.

Our approach is a fully systematic interpretation of control flow information for various fragments as well as nested fragments present in UML 2.x sequence diagram. The control flow information generated from UML 2.x sequence diagram used to handle fragment and nested fragment structure present in sequence diagram, while generating test scenarios. . Subsequently our approach uses these control flow primitives for test scenario generation. Our approach is fully automatic. The test scenarios thus generated are suitable for functional testing and detecting interaction and scenario faults.

REFERENCES

[1] Garousi, Vahid, Lionel C. Briand, and Yvan Labiche. “Control ow analysis of UML 2.0 sequence diagrams." In Model Driven Architecture foundations and Applications, pp. 160-174. Springer Berlin Heidelberg, 2005.

[2] Sarma, Monalisa, Debasish Kundu, and Rajib Mall. “Automatic test case generation from UML sequence diagram." In Advanced Computing and Communications. ADCOM 2007. International Conference on, pp. 60-67. IEEE, 2007.

[3] Cartaxo, Emanuela G., Francisco G. Oliveira Neto, and P. D. Machado. “Test case generation by means of UML sequence diagrams and labeled transition systems." In Systems, Man and Cybernetics. ISIC. IEEE International Conference on, pp. 1292-1297. IEEE, 2007.

[4] Khandai, Monalisha, Arup Abhinna Acharya, and Durga Prasad Mohapatra. “A novel approach of test case gener- ation for concurrent systems using UML Sequence Dia- gram." In Electronics Computer Technology (ICECT), 3rd International Conference on, Vol. 1, pp. 157-161. IEEE, 2011.

[5] D. Kundu, D. Samanta, and R. Mall. "An approach to convert XMI representation of UML 2. x interaction dia- gram into control ow graph". ISRN Software Engineering, pp. 1-22, 2012.
[6] G. Booch, I. Jacobson, and J. Rumbaugh. "The Uni_ed Modeling Language User Guide". Pearson Education.
[7] T. T. Dinh-Trong, S. Ghosh, and R. B. France. "A sys- tematic approach to generate inputs to test UML design models". In Software Reliability Engineering, 2006. IS- SRE'06. 17th International Symposium on, pp. 95-104. IEEE, 2006.
[8] Y. Lei and N. Lin. "Semi automatic test case generation based on sequence diagram". In International Computer Symposium (ICS2008), pp. 349-355, 2008.
[9] Nayak and D. Samanta. "Model-based test cases syn- thesis using UML interaction diagrams". ACM SIGSOFT Software Engineering Notes, 34(2):1-10, 1993.
[10] Rountev, S. Kagan, and J. Sawin. "Coverage criteria for testing of object interactions in sequence diagrams". In Fundamental Approaches to Software Engineering, pp. 289-304. Springer, 2005.
[11] M. Shirole and R. Kumar. "UML behavioral model based test case generation: a survey". ACM SIGSOFT
[12] Samuel, Philip, and Anju Teresa Joseph. "Test sequence generation from UML sequence diagrams." *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on*. IEEE, 2008.

Table 1: Test scenarios generated for Generate Bills use case.

Test Scenario ID	Messages from non user object to user object	Scenario or path sequence
1	M7	Start ->M1 ->M2 ->loopS1->M3->loopE1->loopS1->altS1->M4->altE1 ->altS2 ->M6 -> breakS1-> Expected_output0->M7
2	M7	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6-> breakS1-> Expected_output0->M7
3	M7	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1-> Expected_output0->M7
4	M7	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1-> Expected_output0->M7
5	M10	Startt->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2-> M9-> breakS2-> Expected_output1->M10
6	M10	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9-> breakS2-> Expected_output1->M10
7	M10	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> Expected_output1- >M10
8	M10	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> Expected_output1- >M10
9	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M12->M13->M14->parE1-> Expected_output2->M15
10	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M12->M14->M13->parE1-> Expected_output2->M15
11	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M13->M12->M14->parE1-> Expected_output2->M15
12	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M13->M14->M12->parE1-> Expected_output2->M15
13	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M14->M12->M13->parE1-> Expected_output2->M15
14	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M14->M13->M12->parE1-> Expected_output2->M15
15	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M12->M13->M14->parE1-> Expected_output2->M15
16	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M12->M14->M13->parE1-> Expected_output2->M15
17	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M13->M12->M14->parE1-> Expected_output2->M15
18	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M13->M14->M12->parE1-> Expected_output2->M15
19	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M14->M12->M13->parE1-> Expected_output2->M15
20	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8-> loopS2->M9-> breakS2->loopE2->loopS2-> M11->parS1->M14->M13->M12->parE1-> Expected_output2->M15
21	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2- >M11->parS1->M12->M13-> M14->parE1-> Expected_output2->M15
22	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2- >M11->parS1->M12->M14-> M13->parE1-> Expected_output2->M15

23	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M13->M12-> M14->parE1-> Expected_output2->M15
24	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M13->M14-> M12->parE1-> Expected_output2->M15
25	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M14->M12-> M13->parE1-> Expected_output2->M15
26	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M14->M13-> M12->parE1-> Expected_output2->M15
27	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M13-> M14->parE1-> Expected_output2->M15
28	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M12->M14-> M13->parE1-> Expected_output2->M15
29	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M13->M12-> M14->parE1-> Expected_output2->M15
30	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M13->M14-> M12->parE1-> Expected_output2->M15
31	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M14->M12-> M13->parE1-> Expected_output2->M15
32	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2->M9->breakS2-> loopE2->loopS2->M11->parS1->M14->M13-> M12->parE1-> Expected_output2->M15
33	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M12->M13->M14-> parE1-> Expected_output2->M15
34	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M12->M14->M13-> parE1-> Expected_output2->M15
35	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M13->M12->M14-> parE1-> Expected_output2->M15
36	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M13->M14->M12-> parE1-> Expected_output2->M15
37	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M14->M12->M13-> parE1-> Expected_output2->M15
38	M15	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8-> loopS2->M11->parS1->M14->M13->M12-> parE1-> Expected_output2->M15
39	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M12->M13->M14->parE1-> Expected_output2->M15
40	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M12->M14->M13->parE1-> Expected_output2->M15
41	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M13->M12->M14->parE1-> Expected_output2->M15
42	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M13->M14->M12->parE1-> Expected_output2->M15
43	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M14->M12->M13->parE1-> Expected_output2->M15
44	M15	Start->M1->M2->loopS1->M3->loopE1->loops1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11-> parS1->M14->M13->M12->parE1-> Expected_output2->M15
45	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12-> M13->M14->parE1-> Expected_output2->M15
46	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12-> M14->M13->parE1-> Expected_output2->M15
47	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M13-> M12->M14->parE1-> Expected_output2->M15
48	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M13-> M14->M12->parE1-> Expected_output2->M15
49	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M14-> M12->M13->parE1-> Expected_output2->M15
50	M15	Start->M1->M2->loopS1->altS1->M4->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M14-> M13->M12->parE1-> Expected_output2->M15
51	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12->M13->M14->parE1-> Expected_output2->M15
52	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M12->M14->M13->parE1-> Expected_output2->M15
53	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M13->M12->M14->parE1-> Expected_output2->M15

54	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M13->M14->M12->parE1-> Expected_output2->M15
55	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M14->M12->M13->parE1-> Expected_output2->M15
56	M15	Start->M1->M2->loopS1->altS1->M5->altE1->altS2->M6->breakS1->M8->loopS2-> M11->parS1->M14->M13->M12->parE1-> Expected_output2->M15
57	M16	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M4->altE1->altS2-> Expected_output3->M16
58	M16	Start->M1->M2->loopS1->M3->loopE1->loopS1->altS1->M5->altE1->altS2-> Expected_output3->M16
59	M16	Start->M1->M2->loopS1-> altS1->M4->altE1->altS2-> Expected_output3->M16
60	M16	Start->M1->M2->loopS1-> altS1->M5->altE1->altS2-> Expected_output3->M16