Automatic Test Case Generation from UML Sequence Diagrams

Monalisa Sarma
Department of Computer Science & Engineering
Indian Institute of Technology
Kharagpur WB 721302,
monalisas@cse.iitkgp.ernet.in

Debasish Kundu
School of Information Technology
Indian Institute of Technology
Kharagpur WB 721302,
dkundu@sit.iitkgp.ernet.in

Rajib Mall
Department of Computer Science & Engineering
Indian Institute of Technology
Kharagpur WB 721302,
rajib@cse.iitkgp.ernet.in

Abstract

This paper presents a novel approach of generating test cases from UML design diagrams. Our approach consists of transforming a UML sequence diagram into a graph called the sequence diagram graph (SDG) and augmenting the SDG nodes with different information necessary to compose test vectors. These information are mined from use case templates, class diagrams and data dictionary. The SDG is then traversed to generate test cases. The test cases thus generated are suitable for system testing and to detect interaction and scenario faults.

1. Introduction

With the increasing complexity and size of software applications more emphasis has been placed on object-oriented design strategy to reduce software cost and enhance software usability. However, object-oriented environment for design and implementation of software brings about new issues in software testing. This is because the important features of an object oriented program, such as, encapsulation, inheritance, polymorphism, dynamic binding etc. create several testing problems and bug hazards [3].

Last decade has witnessed a very slow but steady advancement made to the testing of object-oriented systems. One significant approach is the generation of test cases from UML models. The main advantage with this approach is that it can address the challenges posed by object-oriented paradigms. Moreover, test cases can be generated early in the development process and thus it helps in finding out many problems in design if any even before the program is implemented. However, selection of test cases from UML model is one of the most challenging tasks [1]. A test case consists of a test input values, its expected output and the constraints, that is the pre- and post condition for that input values. This information may not be readily available in the design artifacts. As a way out to this problem, several researches propose to augment the design models with testable information prior to the testing process [4]. However, this complicates the automatic test case generation effort.

In this paper we propose an automatic test case generation method using UML [10] models. We use sequence diagram as a source of test case generation. Our generated test suite aims to cover various interaction faults as well as scenario faults. For generating test data, sequence diagram alone may not be enough to decide the different components, i.e. input, expected output and pre- and post- condition of a test case. We propose to collect this information from the use case template, class diagram and data dictionary. These are associated with the use case for which the sequence diagram is considered. We consider OCL 2.0 [13] in our work.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Our approach is discussed in Section 3. Information to be stored in the graph is discussed in Section 4. In Section 5, coverage criteria, generation of test cases from the graph is presented. Implementation of our approach is discussed in Section 6. Finally, Section 7 concludes this paper.

2. Related Work

Several research attempts have been reported on scenario coverage based system testing [5, 6, 11]. These attempts are basically black box approaches and do not take into consideration the structural and behavioral design into consideration. Further, these work [5, 6, 11] require using their proposed custom modeling notations. Frohlich and Link [5] construct a statechart model in which the states are abstractions representing the interval between two successive messages sent to the system by a user. The coverage attempted is transition coverage of the statechart model, which in essence is the coverage of all interactions (message exchanges) of the user with the system. Hartmann et al [6] proposed an automatic test case generation methodology based on the interactions between the system and its user. To model interactions, they semi-automatically convert the textual description

0-7695-3059-1/07 $25.00 © 2007 IEEE
DOI 10.1109/ADCOM.2007.68

of use cases into activity diagrams. Their approach manually annotates the design before test generation. Riebisch et al [11] generate system-level test cases from usage models. A usage model is derived from state diagrams, and is not amenable to full automation.

Briand and Labiche [4] describe the TOTEM (Testing Object orienTed systEms with the unified Modeling language) system testing methodology. System test requirements are derived from early UML analysis artifacts such as, use case diagrams and sequence diagrams associated with each use case and class diagrams. They capture the sequential dependencies between use cases into the form of an activity diagram with the intervention of application domain experts and derive test cases from it. Based on these sequential dependencies, they generate legal sequences of use cases for test case generation. Their approach is, in essence, a semi-automatic way of scenario coverage with pre-specified initial conditions and test oracles.

For testing different aspects of object interaction, several researchers have proposed different technique based on UML interaction diagrams (sequence and collaboration diagram) [2, 7, 8, 12, 16, 17]. Bertolino and Basanieri [16] proposed a method to generate test cases using the UML Use Case and Interaction diagrams (specifically, the Message Sequence diagram). It basically aims at integration testing to verify that the pre-tested system components interact correctly. They use category partition method [18] and generate test cases manually following the the sequences of messages between components over the Sequence Diagram . In another interesting work, Basanieri et al. [8] describe the CowSuite approach which provides a method to derive the test suites and a strategy for test prioritization and selection. This approach construct a graph which is a mapping of the project architecture by analysing the use case diagrams and sequence diagrams. This graph is then traversed using a modified version of the depth-first search algorithm. and use category partition method [18] for generating tests manually. An approach proposed in [7] focuses on real-time systems only. The approach proposed in [12] generates test cases based on UML sequence diagram that are reverse engineered from the code under test.

3. Proposed Approach

Given a sequence diagram (*SD*), we transform it into a graphical representation called sequence diagram graph (*SDG*). Each node in the *SDG* stores necessary information for test case generation. This information are collected from the use case template (also called extended use case), class diagrams, and data dictionary expressed in the form of object constrained language (OCL), which are associated with the use case for which the sequence diagram is considered. We then traverse *SDG* and generate test cases based on a coverage criteria and a fault model. A schematic diagram of our approach is shown in Fig. 1. In the following sections, we discuss the different steps of our approach.

3.1 Transformation of an SD into an SDG

In this section, we first define an *SDG*. Subsequently, we present our methodology to transform a sequence diagram into an *SDG*.

Definition of SDG:

$SDG = \{S_{SDG}, \sum_{SDG}, q0_{SDG}, F_{SDG}\}$, where

S_{SDG} is the set of all nodes representing various states of operation scenarios; Each node basically represents an event.

$\sum_{SDG}$ is the set of edges representing transitions from one state to another.

$q0_{SDG}$ is the initial node representing a state from which an operation begins.

F_{SDG} is the set of final nodes representing states where an operation terminates.

In order to formulate a methodology, we define an operation scenario as a quadruple, *aOpnScn*: <*ScnId*; *StartState*; *MessageSet*; *NextState*>. A unique number called *ScnID* identifies each operation scenario. Here, *StartState* is a starting point of the *ScnId*, that is, where a scenario starts. *MessageSet* denotes the set of all events that occur in an operation scenario. *NextState* is the state that a system enters after the completion of a scenario. This is the end state of an activity or a use case. It may be noted that an *SDG* has a single start state and one or more end state depending on different operation scenarios.

An event in a *MessageSet* is denoted by a tuple, *aEvent*: <*messageName*; *fromObject*; *toObject [/guard]*> where, *messageName* is the name of the message with its signature, *fromObject* is the sender of the message and *toObject* is the receiver of the message and the optional part */guard* is the guard condition subject to which the *aEvent* will take place. An *aEvent* with * indicates it is an iterative event. *aOpnScn* and *aEvent* is illustrated in Example 1.

Example 1. Fig. 2(a) shows a sequence diagram associated with the use case *PIN Authentication* in a usual ATM system. This sequence diagram consists of five operation scenarios as shown in Fig. 2(a). Individual *aOpnScn* of this sequence diagram is shown in Fig. 2(b). Here, s_i (i = 1...10) denotes a state corresponding to a message m_j (j = 1…8) between two objects with a guard condition *c*, if any. The *StartState* for the different scenarios as shown in Fig. 2(b) is *StateX* and the two different *NextStates* are *StateY* (for *scn1…scn4*) and *StateZ* (for *scn5*). An operation starts with a starting state and undergoes a number of intermediate states due to occurrence of various events. For example, in operation scenario *scn*$_1$, we see three transitions: from *StateX* to s_1, s_1 to s_2 and s_2 to *StateY*.

Creation of SDG:

To create the *SDG* for any sequence diagram, we first identify *OpnScn,* the set of all operation scenarios where. *OpnScn* = {*aOpnScn*$_1$, *aOpnScn*$_2$, …, *aOpnScn*$_m$}. For each *aOpnScn*$_i$ ∈ *OpnScn*, we identify set of all *aEvent*. Initially *SDG contains only the start state i.e StartState.* We then add each *aEvent* of all *aOpnScn*$_i$ ∈ *OpnScn,* followed by its corresponding *NextState,* and remove duplicates, if any. The various events in a loop (iteration) are shown with cyclic edge. The *SDG* of *SD* in Fig. 2(a) is shown in Fig. 2(c).

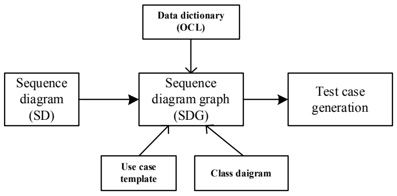
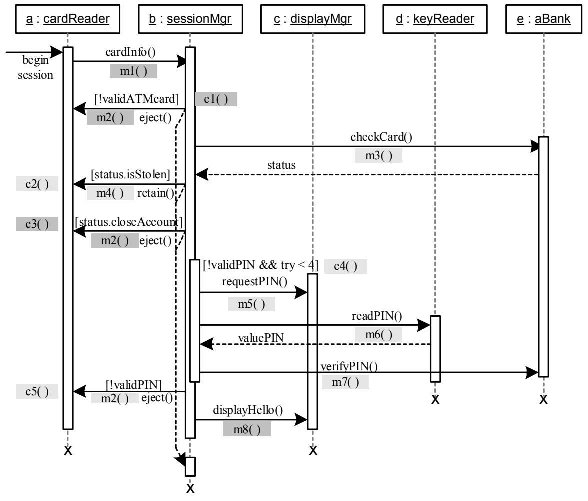
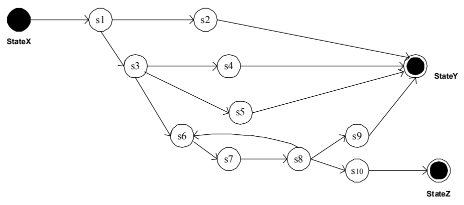


Fig. 1 Schematic representation of our approach



(a) Sequence diagram of PIN Authentication use case in an ATM system



(c) SDG for the sequence diagram in (a)

<scn$_1$	<scn$_2$	<scn$_3$	<scn$_4$	<scn$_5$
StateX	StateX	StateX	StateX	StateX
s1: (m$_1$, a, b)	s1: (m$_1$, a, b)	s1: (m$_1$, a, b)	s1: (m$_1$, a, b)	s1: (m$_1$, a, b)
s2: (m$_2$, b, a) |c1	s3: (m$_3$, b, e)	s3: (m$_3$, b, e)	s3: (m$_3$, b, e)	s3: (m$_3$, b, e)
StateY>	s4: (m$_4$, b, a)|c2	s5: (m$_2$, b, a)|c3	s6: (m$_5$, b, c)|c4*	s6: (m$_5$, b, c)|c4*
	StateY>	StateY>	s7: (m$_6$, b, d)|c4*	s7: (m$_6$, b, d)|c4*
			s8: (m$_7$, b, e)|c4*	s8: (m$_7$, b, e)|c4*
			s9: (m$_2$, b, a)|c5	s10: (m$_8$, b, c)
			StateY>	StateZ>

(b) Five operation scenarios represented in the form of quadruples

Fig. 2 Sequence diagram and its formulation

4. Information to be stored in the SDG

SDG plays an important role in our automatic test case generation scheme. For this, *SDG* contain certain necessary information for test generation. It is evident that each node in the *SDG* is mapped to an interaction with or without a guard between two objects o_i and o_j through a message m_k. Information regarding this needs to be stored in its corresponding node in the *SDG*. The following data needs to be stored: attributes of the corresponding objects at that state, arguments in the method, and predicate of the guard if any, involved in the interaction. This information is collected from the class diagram. In addition to this a node also stores range of values of all attributes of the objects at the state. This information can be obtained from the data dictionary associated to the given design. Further, a node stores the expected results for an occurrence of an event. For example, let us consider a method m_i of an object o_k is invoked by another object o_j, which results in resetting some member elements d_1, d_2, .., d_l, of the object o_k; then all these resultant values of d_1, d_2, .., d_l would be stored in the node. This information is collected from constraints (such as pre- and post-conditions expressed in OCL) specified in the corresponding method in the class diagram and from the use case template. Finally, suppose, the *SDG* under consideration represents three scenarios *scn1*, *scn2*, and *scn3* and i_1, i_2, and i_3 are the set of data which trigger these three scenarios, respectively. Then the *StateX* node should store all possible values for the set of data i_1, i_2, and i_3. These input and corresponding expected outputs are obtained from the use case template.

5. Test Case Generation

A sequence diagram represents various interactions possible among different objects during an operation. A test set is therefore necessary to detect faults if any when an object invokes a method of another object and whether the right sequence of message passing is followed to accomplish an operation. From the *SDG* it is evident that covering all paths from the start node to a final node would eventually cover all interactions as well as all message sequence paths. We follow the coverage criterion stated below to derive the test set.

Coverage criteria: *All sequence diagram message path sequence coverage criterion: Given a test set T and a sequence diagram D, T must cause each sequence of message path exercise at least once.*

To generate test cases that satisfy the criteria, we first enumerate all possible paths from the start node to a final node in the *SDG*. Each path then would be visited to generate test cases. The algorithm to generate test set satisfying the coverage criterion is precisely stated in the *Algorithm TestSetGeneration*.

Every test strategy targets to detect certain categories of faults called the fault model [3]. Our test strategy is based on the following fault model.

Interaction fault: In object-oriented programs, sequences of messages are exchanged among objects to accomplish some operations of interest [3, 9]. Several faults such as incorrect response to a message, correct message passed to a wrong object or incorrect message passed to the right object, message invocation with improper or incorrect arguments, message passed to yet to be instantiated objects, incorrect or missing output etc. may occur in an interaction [9].

Scenario fault: A sequence diagram depicts several operation scenarios. Each scenario corresponds to a different sequence of message path in the sequence diagram. For a given operation scenario, sequence of message may not follow the desired path due to incorrect condition evaluation, abnormal termination etc. [3].

Algorithm *TestSetGeneration*
Input: Sequence diagram graph *SDG*
Output: Test suite T
Steps:

```
1.  Enumerate all paths $P = \{P_1, P_2, ..., P_n\}$ from the
    start node to a final node in the SDG.
2.  For each path $P_i \in P$ do
3.      $n_j = n_x$ // $n_j$ is the current node; start with $n_x$,
                   // the start node
4.      $preC_i$ is the precondition of the scenario
        corresponding to $scn_i$ stored in $n_x$
5.      $t_i \leftarrow \Phi$ // The test case for the scenario $scn_i$,
        initially empty
6.      $n_j = n_y$ // Move to the first node of the scenario
                   // $scn_i$
7.      While $(n_j \neq n_z)$ do     // $n_z$ being a final node
8.          $e_j = \langle m, a, b, c \rangle$ // The event
                   // corresponding to the node $n_j$
                   // and $m(\ldots)$ is invoked with
                   //a set of arguments $a_1, a_2, \ldots, a_l$
9.          If $c = \Lambda$ then //If there is no guard condition
10.             Select test case
                $t = \{preC, I(a_1, a_2, ..., a_l), O(d_1, d_2, ..., d_m), postC\}$
                where $preC$ = precondition of the method $m$
                $I(a_1, a_2, ..., a_l)$= set of input values for the
                method $m(\ldots)$from fromObject
                $O(d_1, d_2, ..., d_m)$= set of resultant values in
                the toObject when the method $m(\ldots)$ is executed
                $postC$ = the postcondition of the method $m(\ldots)$
11.             Add $t$ to the test set $t_i$, that is, $t_i = t_i \cup t$
12.         EndIf
13.         If $c \neq \Lambda$ then //method $m$ is under guard condition
14.             $c(v) = (c_1, c_2, ..., c_l)$ // The set of value of
                   // clauses on the path $P_i$
15.             Select test case
                $t = \{preC, I(a_1, a_2, ..., a_l), O(d_1, d_2, ..., d_m), c(v), postC\}$
                where $preC$ = precondition of the method $m$
                $I(a_1, a_2, ..., a_l)$= set of input values for
                the method $m(\ldots)$ obtained from fromObject
                $O(d_1, d_2, ..., d_m)$ = set of resultant values
      in the *toObject* when the method $m(\ldots)$ is executed
      *postC* = the postcondition of the method $m(\ldots)$
16.            Add $t$ to the test set $t_i$, that is, $t_i = t_i \cup t$
17.            EndIf
18.       $n_j = n_k$ // Move to the next node $n_k$ on the path $P_i$
19.            $T \leftarrow T \cup t_i$
20.       EndWhile
21.       Determine the final output $O_i$ and $postC_i$ for the
          $scn_i$ stored in $n_z$
22.       $t = \{preC_i, I_i, O_i, postC_i\}$
23.    Add the test case $t$ to the test case T, that is,
       $T \leftarrow T \cup t$
24. EndFor
25. Return ($T$)
26. Stop
```

TestSetGeneration starts by enumerating all paths in the *SDG,* from the start node to the different final nodes. Steps 2 to 24 are iterated for each path in the *SDG.* A path essentially corresponds to a scenario. Step 4 determines the initial precondition of the scenario from the start node n_x. For each considered path, Steps 7 to 20 determine the various pre conditions, input, output and post conditions for each interaction of the considered scenario. This gives the test cases for finding out interaction faults if any. And finally Step 22 gives the test case corresponding to the scenario as a whole.

6. Experimental Results

We used MagicDraw v. 10.0 to produce the UML design artifact. This design artifact is exported in XML format. We have written a parser that reads a UML sequence diagram in XMI/XML format and convert it into *SDG*. The nodes of *SDG* is defined as "generic" using template definition in C++, and the structure is decided so that it can dynamically store any number of links and any amount of information. We consider the use case template according to the Extended UC Model in IBM's Rational Rose software [14]. The structure is also similar to the template proposed in [5]. The OCL 2.0 syntax is followed to represent data dictionary. We consider breadth-first traversal algorithm to enumerate all paths in an *SDG*. For the specification of a test case, we consider the test specification language according to the IEEE Standard 829 of TSL [14]. Test cases generated are recorded in a temporary file for future references.

We implement our approach using C++ programming language in Linux OS and run the program in Intel Machine with P-IV processor at 2.6 GHz. A snapshot of a sample output on a run over the example of *ATM PIN Validation* use case is shown in Fig. 3. For brevity, Fig. 3 only shows the test case corresponding to the scenario fault detection. We enumerate 5 different paths on the *SDG* (in Fig. 2(c)) giving 5 test cases as shown in Fig. 3.

7. Conclusions

We focused on automatic generation of test cases from sequence diagrams. A methodology has been proposed to convert the UML sequence diagram into a graph called sequence diagram graph. The information those are required for the specification of input, output, pre- and post- conditions etc. of a test case are retrieved from the extended use cases, data dictionary expressed in OCL 2.0, class diagrams (composed of application domain classes and their contracts) etc. and are stored in the *SDG*. The approach does not require any modification in the UML models or manual intervention to set input/output etc. to compute test cases. Hence, our approach provides a tool that straightway can be used to automate testing process. We follow a graph based methodology and run-time complexity is governed by the breadth-first search algorithm to enumerate all paths, which is $O(n^2)$ in the worst case for a graph of n nodes. This implies that our approach can handle a large design efficiently.

References

[1] A. Abdurazik and J. Offutt, *Generating Tests from UML Specifications*, in the Proceedings of 2nd International Conference on Unified Modeling Language (UML'99), Fort Collins, CO, 1999.

[2] A. Abdurazik, and J. Offutt, *Using UML Collaboration diagrams for static checking and test generation*, in: Proceedings of the Third International Conference on the UML, Lecture Notes in Computer Science, Springer-Verlag GmbH, York, UK, vol. 939, 2000, pp. 383–395.

[3] R. V. Binder, *Testing Object-Oriented System Models, Patterns, and Tools*, Addison-Wesley, NY, 1999

[4] L. Briand and Y. Labiche, *A UML-Based Approach to System Testing*, in the Journal of Software and Systems Modeling, Springer Verlag, Vol. 1, pp. 10-42, 2002.

[5] P. Fröhlick and J. Link, *Automated Test cases Generation from Dynamic Models*, in the Proceedings of the European Conference on Object-Oriented Programming, Springer Verlag, LNCS 1850, pp. 472-491, 2000.

[6] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, *A UML-based Approach to System Testing*, Journal of Innovations System Software Engineering, Vol. 1, PP. 12-24, 2005.

[7] M. Lettrari M. and J. Klose, *Scenario-Based Monitoring and Testing of Real Time UML Models*, in the Proceedings of UML 2001, Springer Verlag, pp. 312-328.

[8] F. Basanieri, A. Bertolino, and E. Marchetti, *The cow suit approach to planning and deriving test suites in UML projects*, in Proceedings of the 5th International Conference on the UML, LNCS, 2460, 2002, pp. 383–397.

[9] J. D. McGregor, and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, Addison Wesley, NJ, 2001.

[10] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell*, O'Reilly, NY, USA, 2005.

[11] M. Riebisch, I. Philippow, and M. Gotze, *UML-Based Statistical Test Case Generation*, in the Proceeding of ECOOP 2003, Springer Verlag, LNCS 2591, pp. 394-411, 2003.

[12] P. Tonella, and Potrich, A. *Reverse Engineering of the Interaction Diagrams from C++ Code*, in the Proceedings of

IEEE International Conference on Software Maintenance (2003) 159–168.

[13] *Object Constraint Language 2.0* is available from Object Mangement Group's web site (http://www.omg.org/).

[14] A Rational Approach to Software Development using Rational Rose 4.0, IBM's Rational Rose, (http://www.ibm.com/)

[15] G.H. Walton, and J.H. Poore, *Statistical testing of software based on usage mode*l, Software-Practice and Experience, Vol 25, 1995, pp 97-108.

[16] A. Bertolino, and F. Basanieri, *A practical approach to UML-based derivation of integration tests*, in Proceedings of the Fourth International Software Quality Week Europe, Brussels, Belgium, 2000.

[17] F. Fraikin, and T. Leonhardt, *SEDITEC-testing based on sequence diagrams*, in Proceedings 17th IEEE International Conference on ASE, 2002, pp. 261–266.

[18] T.J. Ostrand, and M.J. Balcer, *The category-partition method for specifying and generating fuctional tests*, Communications of the ACM 31 (6) (1998).

```
 1.  Test name = "ATM PIN Validation"
 2.  Preconditions:   ATM   is   idle   and   displaying   a   welcome
                       message. User inserts a card
 3.  Test case: Scenario 1
 4.      Input: Card = "Not ATM"
 5.      Output: Eject card
 6.      Postcondition: Displays welcome message
 7.  Test case: Scenario 2
 8.      Input: Card = "ATM", Status = "Stolen"
 9.      Output: Eject card
10.      Postcondition: Back to the initial state
11.  Test case: Scenario 3
12.      Input: Card = "ATM", Status = "Okay", Account = "Close"
13.      Output: Eject card
14.      Postcondition: Displays welcome message
15.  Test case: Scenario 4
16.      Input: Card = "ATM", Status = "Okay",
         Account = "Open", PIN = "Invalid"
17.      Output: Message "Invalid PIN: Try Again"
18.      Postcondition: Displays welcome message
19.      Input: Card = "ATM", Status = "Okay",
         Account = "Open", PIN = "Invalid"
20.      Output: Message "Invalid PIN: Try Again"
21.      Postcondition: Displays welcome message
22.      Input: Card = "ATM", Status = "Okay",
         Account = "Open", PIN = "Invalid"
23.      Output: Message "Invalid PIN: Try Again"
24.      Postcondition: Displays welcome message
25.      Input: Card = "ATM", Status = "Okay",
                Account = "Open". PIN = "Invalid", Try = <4>
26.      Output: Message "Invalid PIN: Try Later"
                 Eject card
27.      Postcondition: Displays welcome message
28.  Test case: Scenario 5
29.      Input: Card = "ATM", Status = "Okay",
                Account = "Open", PIN = "Valid"
30.      Output: Display "Hello"
31.      Postcondition: Eject card
32.  Postcondition: Display menu for transaction
```

Fig. 3 Snapshot of test run with the sequence diagram of ATM PIN Validation