Automatic Test Case Generation from UML Models

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

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

Abstract

This paper presents a novel approach of generating test cases from UML design diagrams. We consider use case and sequence diagram in our test case generation scheme. Our approach consists of transforming a UML use case diagram into a graph called use case diagram graph (UDG) and sequence diagram into a graph called the sequence diagram graph (SDG) and then integrating UDG and SDG to form the System Testing Graph (STG). The STG is then traversed to generate test cases. The test cases thus generated are suitable for system testing and to detect operational, use case dependency, interaction and scenario faults.

Keywords: *Software testing, UML models, Object-oriented system*

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. Most reported research propose test case generation based on program source code. However, generating test cases from program source code, especially for the present day complex applications is very difficult and ineffective. The reason being that the design aspects are very difficult to extract from the code. 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].

In this paper we have proposed an automatic test case generation method using UML [10] models. We consider use case and sequence diagrams as a source of test case generation. Our generated test suite aims to cover operational and use case dependency faults, various interaction as well as scenario faults. For generating the different components of a test case, i.e. input, expected output and pre- and post- condition we also use class diagram and data dictionary along with use case and sequence diagrams. 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. In Section 4, coverage criteria, generation of test cases from the graph is presented. Implementation of our approach is discussed in Section 5. Finally, Section 6 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. Fröhlick 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 of use cases into activity diagrams. Their approach manually annotates the design before the test case generation. Riebisch et al. [11] generate system-level test cases from usage models.

Briand and Labiche [4] describe the TOTEM (Testing Object orienTed systEms with the unified Modeling language) system testing methodology.

0-7695-3068-0/07 $25.00 © 2007 IEEE
DOI 10.1109/ICIT.2007.26

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 sequences of messages between components over the sequence diagram. In another 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 constructs a graph which is a mapping of the project architecture by analyzing 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

In our proposed approach we convert a system under test into a graph called *System Testing Graph* (STG), which is an integration of use case and sequence diagram. We first transform an use case diagram (UD) into a use case diagram graph (UDG), sequence diagram (SD) into a sequence diagram graph (SDG) and then integrate UDG and SDG to form STG. Information necessary to derive test cases is pre-stored into this graph. These information are retrieved 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 UML diagrams. The graph so obtained is then traversed to generate test cases automatically based on a coverage criteria and a fault model. In the following sections, we discuss the different steps of our approach.

3.1 Transformation of an UD into an UDG

In this section, we first define an *UDG*. Subsequently, we present our methodology to transform a UD into an *UDG*.

Definition of UDG:

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

$S_{UDG} = UC \cup A$ where $UC = \{U_1, U_2, ..., U_i\}$ is a finite set of nodes representing use cases. $A = \{A_1, A_2, ..., A_j\}$ is a finite set of nodes representing actors.

$\sum_{UDG} = AU \cup UD$, where $AU = \{A \times U\} \cup \{U \times A\}$ is a set of associations between an actor $A_i \in A$ and a use case $U_i \in U$, $UD = \{U \times U\}$ represents use case dependency relationships between two use cases $U_i, U_j \in U$.

$q0_{UDG} \in \{A_1, A_2, ..., A_i\}$ is the set of start node representing those actors that act as data source such that $q0_{UDG} \times U \in AU$.

$F_{UDG} \in \{A_1, A_2, ..., A_j\}$ is the set of final nodes representing those actors that act as data sinks such that $U \times F_{UDG} \in \{U \times A\}$.

Now, we discuss the transformation of a UCD into a UDG. A use case in UCD can be mapped to a node in UDG. All actors would be mapped to either a start node or a final node or both. A directed edge from a node U_i to U_j is used to represent the sequential dependency of U_j on U_i.

From the definition of UDG, it may be noted that for some use case diagrams no actor may correspond to a final node and in that case the set F_{UDG} is a null set. Further, an actor may belong to both the sets $q0_{UDG}$ and F_{UDG} and hence a node in UDG may be both a start node as well as a final node. Fig. 1(a) shows the UD of an *online purchase* system and the corresponding *UDG* is shown in Fig. 1(b).

The following information are required to be stored in nodes in an *UDG*.

- Data from an actor to a use case or from a use case to an actor.
- Pre- and post- conditions of a use case.

3.2 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.

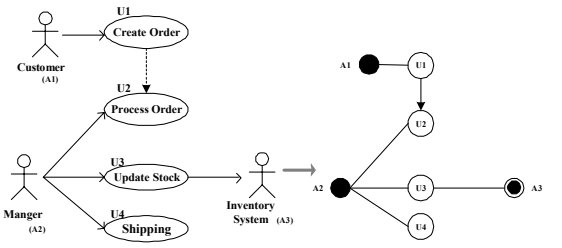
F_{SDG} is the set of final nodes representing states where an operation terminates.

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 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.

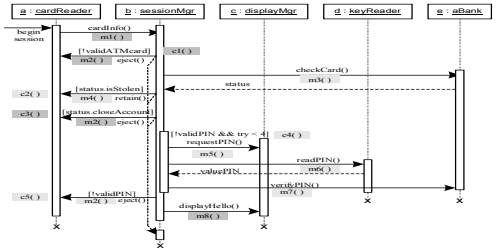
Fig. 2(a) shows a sequence diagram associated with the use case *PIN Authentication* in a usual ATM system amd its five scenarios is shown in Fig. 2(c). The *SDG* of *SD* in Fig. 2(a) is shown in Fig. 2(b).

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 with the given design. Further, a node stores the expected results for an occurrence of an event.

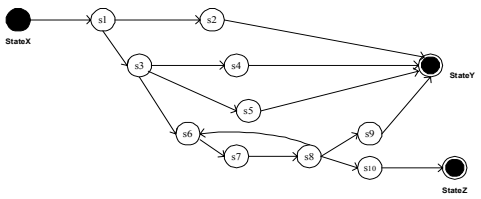


(a) A use case diagram (*UCD*) (b) Use case diagram graph (*UDG*)

Fig. 1 Creating a UDG from a UCD



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



(b) SDG of the sequence diagram in

<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 StateY>	s10: (m_8, b, c) StateZ>

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

Fig. 2 Illustration of creating SDG

3.3 Integration of UDG and SDG into STG

After the creation of UDG and SDG the next step is to integrate these two graphs into a single graph called the system testing graph (STG). In the following, we define an STG.

Definition of STG:

$STG =< S, \sum, \delta, q_0, F >$ where

$S = S_{UDG} \cup S_{SDG}$ is the set of all nodes in the STG

$\sum = \sum_{UDG} \cup \sum_{SDG} \cup \sum_{G}$ where

$\sum_{G} = (S_{UDG} \times q0_{SDG})$ denotes connectivity between *UDG* to *SDG*s

$q0_{STG} = q0_{UDG}$ is the set of start nodes in the *STG*.

$F_{STG} = F_{SDG} \cup F_{UDG}$ is the set of final nodes in the *STG*.

Starting with an *UDG,* we integrate the *SDG*s into it following the definition of *STG* as mentioned above. A link from the use case node to the start node of the corresponding *SDG* is maintained.

4. Test Case Generation

Having stored all essential information for test generation in the STG, we now traverse the STG to generate test cases. We design an algorithm *TestSuiteGeneration* to automatically traverse the STG so as to generate test cases in accordance with a coverage criterian. The *TestSuiteGeneration* traverse the STG at two levels. The traversal begins with the UDG. We term this traversal as Level 1 traversal. This traversal visits all use cases and generates test cases for detecting initialization faults. At Level 2 traversal, starting from a use case node the corresponding SDG is visited and test cases are generated to detect operational faults. Finally, we enumerate all paths in the UDG to identify all use case dependency and generate test case to detect use case dependency faults. We propose the following two coverage criteria to generate test cases.

Coverage criteria (C1): *All use case and all use case dependency relations criterion: Given a test set T and a use case diagram D, T must cause each use case and each dependency path to be exercised at least once.*

Coverage criteria (C2): *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 to be exercised at least once.*

The algorithms *TestSuiteGenerationUDG* and *TestSetGenerationforSDG* satisfying the coverage criteria C1 and C2, respectively is stated below.

Algorithm *TestSuiteGeneration*
Input: System testing graph *STG*
Output: Test suite *T*

1. $T \leftarrow \Phi$
   /* Find all *initialization faults* and *operational faults* */
2. *Call TestSetGenerationUDG(UDG)*
   /* Find all *use case dependency faults* */
3. *P = EnumerateAllPaths (UDG)*
4. For each path $p_i \in P$ do
5. *UD = FindAllUseCaseDependency*(*UDG*)
6. For each $UD_i \in UD$ do
7. For each $U_j \in UD_i$ do
8. *AUD = FindActorToUC*(U_i) // The set of actor-
   // to-use case association in the use case U_j
9. For each $au_k \in AUD$ do
10. *I = GetInputDomain*(au_k)
11. *O = GetOutputDomain*(au_k)
12. *preC* = pre-condition of U_j
13. *postC* = post-Condition of U_j
14. $t = SelectTestCase\{I, O, preC, postC\}$
15. $T \leftarrow T \cup t$
16. EndFor
17. EndFor
18. EndFor
19. Stop

4.1 Test Case Generation from UDG

Every use case is implemented as the collaborative actions of several objects. However, for a use case to successfully execute, the objects must be in certain desired states for the use case to start. In other words, proper context must exist for a use case to execute as per specification and produce correct results. Further, a use case specifies a set of responses to be produced for some specific combinations of external inputs and system state. A system leads to an operational fault if each use case does not obey the desired input-output relationships. It may also be noted that use cases often have sequential dependencies among each other [4]. In other words, to accomplish a task some use cases need to be executed before the others can execute. In this case, a use case may produce some intermediate results necessary for the successful execution(s) of subsequent use case(s). For example, in an on-line purchase system, an order should be created through a *Create Order* use case before processing a *Process Order* use case. Errors may occur when a use case begins its execution without satisfying the required dependency. A test set is therefore necessary to detect these above mentioned faults if any. We follow the coverage criterion (C1) stated above to derive the test set.

Algorithm *TestSetGenerationUDG*
Input: Use case diagram graph *UDG*
Output: Test set T_1

1. For each use case $U_i \in UDG$ do
2. O = *FindObjects*(U_i) //Find all objects involved in U_i
3. For each $o_{ij} \in O$ do
4. $M = FindAttributes(o_{ij})$
5. *I = IdentifyInputDomain(M)*
6. *O = IdentifyOutputDomain* (*I*, *M*)

7. $t = SelectTestCase(i,o)$ // $i \in I$, $o \in O$
// Generate test case for detecting *initialization faults*
8. $T \leftarrow T \cup t$
9. EndFor
10. $\alpha = U_i \rightarrow SDG$ //Denotes a link from U_i to its *SDG*
11. $T_1 = TestSetGenerationForSDG(\alpha)$
// Generate test case for detecting *operational faults*
12. $T = T \cup T_1$
13. EndFor
14. Stop

4.2. Test Case Generation from SDG

A sequence diagram represents various interactions possible among different objects during an operation. 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]. Further, 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]. 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 (C2) stated above to derive the test set.

To generate test cases that satisfy the criterion C2, we first enumerate all possible paths from the start node to a final node in the *SDG*. Each path then is visited to generate test cases. The algorithm to generate test set satisfying the coverage criterion is stated in the *Algorithm TestSetGenerationForSDG*.

5. Implementation of our approach

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 use case and sequence diagram in XMI/XML format and convert it into UDG and *SDG respectively*. The nodes of UDG and *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. 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.

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

1. P = *EnumerateAllPaths*(SDG)
2. For each path $P_i \in P$ do
3. $n_j = n_x$ *// start with n_x, the start node*
4. $preC_i =$ *FindPreCond* (n_x)
5. $t_i \leftarrow \Phi$ *// The test case for the scenario scn_i*
6. For each node n_j of path P_i do
7. $e_j =$ *FindEvent* (n_j) *// The event //corresponding to the node n_j*
4. If $c = \Lambda$ *//If there is no guard condition*
5. $t = \{preC, I(a_1, a_2, ..., a_l), O(d_1, d_2, ..., d_m), postC\}$
// preC = precondition of the method m
// $I(a_1, a_2, ..., a_l)$ = set of input values for the method m(...) in fromObject
// $O(d_1, d_2, ..., d_m)$ = set of resultant values in the toObject when the method m(...) is executed
// postC = the postcondition of the method m(...)
6. EndIf
7. If $c \neq \Lambda$ then
8. $c(v) = (c_1, c_2, ..., c_l)$ *// The set of value of // clauses on the path P_i*
9. $t = \{preC, I(a_1, a_2, ..., a_l), O(d_1, d_2, ..., d_m), c(v), postC\}$
10. Endif
11. $t_i = t_i \cup t$
12. EndFor
13. $T \leftarrow T \cup t_i$
14. Return (T)
15. Stop

6. Conclusions

We have presented a novel approach of generating test cases from UML design artifacts namely, use case and sequence diagrams. We convert the models into an intermediate representation called system testing graph, which is an integration of intermediate representation of use case and sequence diagrams. Integration of these representations is helpful for the following reasons. Our approach covers three important faults, which usually occur in a system: use case initialization faults, use case dependency faults and operational faults. The first two category of faults can be covered from the UDG, whereas the later from the SDG. It may be noted that SDG models the

operational details of a use case. Hence, if a use case initialization fault occurs then it is imperative to assume faults in its operations and therefore no need to apply test cases corresponding to the operation, that is, those test cases that are derived from the SDG of the use case. Same is true to check the dependency faults where a use case is preceding another use case. The integration will help us to guide whether a test driver needs to apply a specific test suite or not. Another, important reason of integrating is that test data those are necessary for test case are mined once and used in different level such as, use case (to test initialization faults and dependency faults), sequence diagram (to test operational fault) etc. Otherwise, we have to mine same data repeatedly, if they are considered independently. In fact deciding test data, which are embedded in design artifacts is computationally intensive task and our approach significantly able to score in this issue.

References

[1] Abdurazik A. and Ofutt J., *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 A., Offutt J., *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] Binder R. V. , *Testing Object-Oriented System Models, Patterns, and Tools*, Addison-Wesley, NY, 1999

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

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

[7] Lettrari M. and Klose J., *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, E. Marchetti, The cow suit approach to planning and deriving test suites in UML projects Proceedings of the Fifth International Conference on the UML, LNCS, 2460, Springer-Verlag GmbH, Dresden, Germany, 2002, pp. 383–397.

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

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

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

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

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

[17] F. Fraikin, T. Leonhardt, SEDITEC-testing based on sequence diagrams, in: Proceedings 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261–266.

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