p-2689

USING UML DIAGRAMS FOR GENERATING AUTOMATIC TEST CASES

by

G.S.NANDAKUMAR

Reg.No.71206805003

of

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE – 641006

A PROJECT REPORT

Submitted to the

FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements

for the award of the degree

of

MASTER OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

JULY 2009

BONAFIDE CERTIFICATE

Certified that this project report entitled "**USING UML DIAGRAMS FOR GENERATING AUTOMATIC TEST CASES**" is the bonafide work of **G.S.NANDAKUMAR**, who carried out the research under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate

[illegible]

GUIDE

[illegible]

HEAD OF THE DEPARTMENT

The candidate with **University Register No. 71206805003** was examined by us in the project viva – voce examination held on 06/07/2009.

[illegible] 6/7/09

INTERNAL EXAMINER

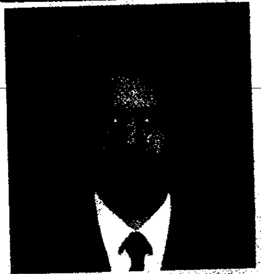
[illegible] 6/7

EXTERNAL EXAMINER

Approved by AICTE and Affiliated to Anna University
(An ISO 9001:2000 Certified Institution)

Department of Information Technology
&
Society for Information Sciences and Computing Technology
Third National Conference on

NETWORKS,INTELLIGENCE AND COMPUTING SYSTEMS



his is to certify that Dr./Prof./Mr/Ms........G.S.Nandakumar M.E CSE..........

Kumaraguru College of Technology.........has Presented a technical paper

d.................Using UML Diagrams for Generating Automatic Test Cases.................

e National Conference NCNICS held on 16th February 2009.

[illegible] Convenor

[illegible] Principal
Dr.V.P.Arunachalam

[illegible] Director cum Secretary
Dr.S.N.Subbramanian

ABSTRACT

Due to the increasing complexity and size of software applications, more emphasis is being placed on object oriented design strategy to reduce software cost and enhance software usability. The usage of Object Orientation for design and implementation brings about new issues in software testing. One significant step is to generate test cases from UML models. This helps to generate test cases early in the development process which can be used to find out many problems in the design phase even before the coding is initiated. This reduces the cost of software development. This work presents an approach to generate test cases from UML Sequence Diagrams and has been implemented in C.

A Sequence Diagram (SD) shows how a sequence of message is intended to service a single user input or an external event. The Sequence Diagram is transformed into a graph known as Sequence Diagram Graph (SDG). The nodes in the SDG are populated with necessary data from Class Diagrams, Use Case Diagrams & Data Dictionary. The SDG is then traversed to generate test cases which can be used to detect scenario faults and faulty interaction among objects.

ஆய்வுச்சுருக்கம்

மென்பொருள் பயன்பாடுகளின் அளவும், கடினத்தன்மையும் அதிகரித்துக் கொண்டு வரும் சூழல் பொருள் சார்ந்த கட்டமைப்புகளின் மீதான கவனத்தை அதிகரிக்கின்றது. இதன் மூலமாக மென்பொருள்களின் விலையைக்குறைக்கவும் மற்றும் உபயோகத்தன்மை ஆகியவற்றை உயர்த்தவும் முடிகிறது.

மென்பொருள்களை வடிவமைப்பதற்காக பொருள் சார்பு தன்மையை உபயோகிப்பது என்பது புதிய வழக்கீடுகளை மென்பொருள்களை பரிசோதிப்பதற்காக முன் வைக்கின்றது. அதில் ஒரு குறிப்பிடத்தக்க வழிமுறையாக இருப்பது சோதனை நிலைகளை UML மாதிரிகளை கொண்டு உருவாக்குவது என்பதாகும். இது சோதனை நிலைகளை கட்டமைப்புப்பணியில் முன்னதாகவே உருவாக்குவதற்கும் அதன் மூலமாக பல தீர்வு காண வேண்டிய பிரச்சினைகளுக்கு வடிவமைப்பின் போதே அவற்றின் வரைவு தொகுப்புகளைத் துவங்கும்முன் தீர்வுகாண உதவுகிறது. இது மென்பொருள் கூட்டமைப்புப்பணியில் செலவுகளைக் குறிப்பிடத்தக்க அளவு குறைக்கின்றது.

தொடர்படங்கள், எப்படி ஒரு தொடர் தகவலானது தனி பயனாளிகளின் உள்ளீட்டிற்கோ அல்லது ஒரு புற நிகழ்வுக்கோ சேவை செய்கின்றது என்பதைக் காட்டுகின்றது. தொடர்படங்கள் வரைபடங்களாக மாற்றப்பட்டு பின்னர் தொடர் வரைபடங்கள் என்றழைக்கப்படுகின்றன. தொடர் வரைபடங்களின் கணுக்கள், வகுப்புப் படங்கள், பயன்நிலைப் படங்கள் மற்றும் தகவல் பொருளடக்கம் இவைகளிலிருந்து பெறப்படும் தேவையான தகவல்கள் மூலம் உருவாக்கப்படுகின்றன. மேலும் இந்தத் தொடர் வரைபடங்கள் பரிசோதனை நிலைகளை உருவாக்குகின்றன. இது பரிமாற்றம் மற்றும் நிலை பிழைகளைக் கண்டறியப்பயன்படுகிறது.

ACKNOWLEDGEMENT

I sincerely thank our Chairman **Arulselvar Dr. N. Mahalingam**, Vice Chairman **Dr. K. Arumugam** & Correspondent **Mr. M.Balasubramaniam** for giving me an opportunity to pursue the M.E course.

I thank, **Prof. R. Annamalai,** Vice Principal, Kumaraguru College of Technology, Coimbatore for providing me with the necessary facility to work on this project.

I wholeheartedly express my deep sense of gratitude to my mentor and guide **Dr. S. Thangasamy,** Dean, Department of Computer Science and Engineering for his constant source of inspiration and guidance.

I thank **Ms. V. Vanitha,** Assistant Professor & Project Coordinator, Department of Computer Science and Engineering for her valuable suggestions and guidance.

My sincere thanks to the Faculty of the Department and other friends for their help and support throughout the course.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS	
SD	Sequence Diagram
SDG	Sequence Diagram Graph
OCL	Object Constraint Language
UML	Unified Modeling Language
RUP	Rational Unified Process

CHAPTER 1

INTRODUCTION

This chapter gives an introduction to software testing, the artefacts of testing, different types of testing and UML.

The purpose of software engineering is to build high quality software. Testing is the most important way of assuring (or controlling) the quality of software. Good practices throughout the development process contribute to the quality of the final product, but only testing can demonstrate that quality has been achieved and identify the problems and the risks that remain. The objective of software testing is to find problems and fix them to improve quality. Software testing typically represents 40% of a software development budget.

Software testing is a predominant role in software development life cycle. Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs. A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. Information derived from software testing may be used to correct the process by which the software is developed.

Software organizations spend considerable portion of their budget in testing related activities. Software testing is important to reveal errors in the software and to ensure that it fulfills its requirements. A well tested software system will be validated by the customer before acceptance. The effectiveness of this verification and validation process depends upon the number of errors found and rectified before releasing the system. This in turn depends upon the quality of test cases generated. Software testing is conducted not only to check if the software meets the functional, technical and security requirements but also to break the software with negative inputs or by incorrect usage.

1.1 TESTING ARTIFACTS

Software testing process can produce several artifacts which are given below.

1.1.1 Test Case

A test case in software engineering normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. This can be as pragmatic as 'for condition x the derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, the user may also be able to see past test results and who generated the results and the system configuration used to generate those results. These past results would usually be stored in a separate table.

A test case is a description of a test, independent of the way a given system is designed. Test cases can be mapped directly to, and derived from Use Cases. Test cases can also be derived from system requirements. One of the advantages of producing test cases from specifications and design is that they can be created earlier in the development life cycle and be ready for use before the programs are constructed. Additionally, when the test cases are generated early, Software Engineers can often find inconsistencies and ambiguities in the requirements specification and design documents. This will definitely bring down the cost of building the software systems as errors are eliminated early during the life cycle.

1.1.2 Test Script

The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools. Today, test scripts can be manual, automated, or a combination of both.

1.1.3 Test Data

The most common testing done is retesting and regression testing, In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

1.1.4 Test Suite

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

1.1.5 Test Plan

A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to the developers. This makes the developers more cautious when developing their code. This ensures that the developers code is not passed through any surprise test case or test plans.

1.2 METHODS OF SOFTWARE TESTING

There are two basic methods of performing software testing:

1. Manual testing
2. Automated testing

1.2.1 Manual Testing

Manual software testing is the process of an individual or individuals manually testing software. This can take the form of navigating user interfaces, submitting information, or even trying to hack the software or underlying database. As one might presume, manual software testing is labor-intensive and slow. There are some things for which manual software testing is appropriate, including:

- User interface or Usability testing.
- Exploratory / ad hoc testing (where testers do not follow a 'script', but rather testers 'explore' the application and use their instincts to find bugs).

- Testing the areas of the application which experience a lot of change.
- User acceptance testing (often, this can also be automated).

The time commitment involved with manual software testing is one of its most significant drawbacks. The time needed to fully test the system will typically range from weeks to months. Manual test processes are often tedious and error prone, and fail to provide the required level of quality. Variability of results depending on who is performing the tests can also be a problem. For these reasons, many companies look for automation as a means of accelerating the software testing process while minimizing the variability of results. In practice, 80 to 95% of tests should be able to be automated. To cut down cost of manual testing and to increase reliability of it, researchers have tried to automate it. One of the important activity in testing environment is automatic test case generation - description of a test, independent of the way a given software system is.

1.2.2 Automated Software Testing

Automated software testing is the process of creating test scripts that can then be run automatically, repetitively, and through many iterations. Done properly, automated software testing can help to minimize the variability of results, speed up the testing process, increase test coverage (the number of different things tested), and ultimately provide greater confidence in the quality of the software being tested. Automated testing is best used for tests which are explicit and repetitive. In today's environment, where resources are limited but quality software is in high demand, one cannot compromise on quality. The results of the automatic tests are seen as a measure of the current quality of the software.

The Software Testing Life Cycle is the road map to automation success. It consists of a set of phases that define what testing activities to do and when to do them. It also enables communication and synchronization between the various groups that have input to the overall testing process.

The advantages of automated testing are given below:

1 Low Running Cost: Running an automated test script before each release of a new version, patch or bug fix is a lot cheaper than a manual test.

2 Better Quality: Especially for individual developers and small companies who would not employ a tester and will perform all testing themselves.

3 Consistency: The test script will perform the same checks every time it is run. A manual test will be affected by human error and it will tend to skip certain areas believed to be stable.

4 Speed: A script will execute many times faster than a manual test, giving a full report on the quality of the product in a few minutes.

5 Compactness: A full compatibility check will be performed by simply copying the application together with the test scripts on all the platforms where it should work. It can give the confirmation that all functionality works indeed as expected.

Automated tests are not meant to completely replace manual testing. They cannot answer questions regarding the program's ease-of-use or user experience, and they cannot be used on small components during development. However they are far superior when it comes to Regression and Functional Testing.

For these reasons, it is of critical importance for the software industry to develop efficient automated methods to achieve higher product quality at lower testing costs.

1.3 UNIFIED MODELING LANGUAGE (UML)

UML is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create abstract models of specific systems. It is a set of techniques for specification, visualization and documentation. The language is based primarily upon object oriented methodology. UML provides capability to explore static and dynamic behavior and physical deployment of a system. UML has become the de facto language of software development.

A very specific benefit of using UML is that it provides a powerful communication vehicle. It is an industry standard, managed by a third party organization. Testers can work on UML diagrams with minimal introduction. The use of UML also allows testers to participate early in the software development process. It provides the ability to improve the test coverage by mapping typical UML assets like Use Cases, Class Diagrams, and Sequence Diagrams to test activities like test cases, test designs, and test implementations.

In UML specification, requirements analysis and testing are usually done using diagrams.UML 2.0 has 13 types of diagrams divided into three categories: Six diagram types represent the structure application, three represent general types of behavior, and four represent different aspects of interactions. These diagrams can be categorized as shown below:

i) Structure Diagrams

Structure diagrams emphasize what things must be in the system being modeled:

1 Class Diagram describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.

2 Component Diagram depicts how a software system is split up into components and shows the dependencies among these components.

3 Composite Structure Diagram describes the internal structure of a class and the collaborations that this structure makes possible.

4 Deployment Diagram serves to model the hardware used in system implementations, the components deployed on the hardware, and the associations among those components.

5 Object Diagram shows a complete or partial view of the structure of a modeled system at a specific time.

6 Package Diagram depicts how a system is split up into logical groupings by showing the dependencies among these groupings.

ii) Behavior Diagrams

Behavior diagrams emphasize what must happen in the system being modeled:

1 Activity Diagram represents the business and operational step-by-step workflows of components in a system. An activity diagram

shows the overall flow of control.

2 State Diagram is a standardized notation to describe many systems, from computer programs to business processes.

3 Use Case Diagram shows the functionality provided by a system in terms of actors, their goals represented as Use Cases, and any dependencies among those Use Cases.

iii) Interaction Diagrams

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

1 Communication Diagram shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.

2 Interaction Overview Diagram a type of activity diagram in which the nodes represent interaction diagrams.

3 Sequence Diagram shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.

4 Timing Diagram is a specific type of interaction diagram, where the focus is on timing constraints.

1.4 ADVANTAGES OF USING UML FOR TESTING

One fundamental problem in the current software development environment is that testing is not begun early enough in the development process. Also generating test cases from program source code for the complex applications is very difficult and ineffective. The reason is that design aspects are very difficult to extract from the code. One significant approach is the generation of test cases from UML models. UML models are an importance source of information for test design. The main advantage with this approach is that it can address the challenges posed by object-oriented paradigms such as encapsulation, inheritance, polymorphism, dynamic binding etc, which create several testing problems and bug hazards. 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. This early defect discovery occurs as the specification is exposed to early scrutiny. The defects detected at an early stage of the development are much cheaper to fix.

Communication problems between developers and testers are notorious for delaying and derailing projects. Since UML is a powerful communication vehicle, testers can work on UML diagrams and derive benefit from them. The use of UML allows testers to participate early in the software development process. Test teams can use UML diagrams and extensions to describe their requirements. Test teams can help architects and developers design test points into their applications, which will trim the test cycle bottleneck and speed up the overall project. Another significant, benefit of using UML is that it gives the ability to improve test coverage by mapping typical UML assets like Use Cases, Class Diagrams, and Sequence Diagrams to test activities like test cases, test designs, and test implementations.

Use Case Diagrams and Sequence Diagrams can be used as a source of test case generation.

1.4.1 Use Cases

Use cases are a popular mechanism for capturing functional and business requirements. Use cases are developed based on the user's perspective since the user is going to use the system. Use cases are a good means to communicate with a customer. They are the unit of work in incremental object oriented software processes like the Rational Unified Process (RUP). They are a good basis for systematic testing. Rumbaugh et al defines uses cases as "The specification of sequence of actions, including variant sequences and error sequences that a system, subsystem or class can perform by interacting with outside actors". In Use Case Diagram, two important factors are used to describe the requirements of the system. They are the actors and Use Cases. Actors are the external entities that interact with the system and Use Cases are the behavior of the system. The Use Cases are used to define the requirements of the system. Mostly each Use Case is converted into a function representing the task of the system. Each of the Use Case can be converted into one or many test cases. In order to make sure that the system does the requirements as it is supposed to do, the test case have to designed according to the tester's perspective.

1.4.2 Sequence Diagrams

Sequence Diagrams are used to model the logic of usage scenarios. A usage scenario is exactly what its name indicates - the description of a potential way the system is used. Sequence Diagrams (Interaction diagrams) are used to formalize single scenarios contained in Use Cases. A Sequence Diagram shows what the actor is doing, what components he is interacting with, and what parameters are getting passed.

They are mainly used to describe dynamic system properties by means of messages and corresponding responses of interacting system components. They describe in detail the activities for Use Cases. So they can also be applied for generating the correct test cases. Sequence Diagrams are a great way to understand and to explain to other people how the objects in a system interrelate as a function of time. A Sequence Diagram unfolds message paths in the time dimension, which provides a useful representation for conceptualizing how collaboration will be accomplished. The utility of Sequence Diagrams is incredible for at least three reasons:

1 Allows developers to really understand the time-sequenced interaction between system objects.

2 Allows technically oriented customers to get a better understanding of how the system will function.

3 Allows developers to identify" failure points" and facilitates setting up test cases.

Sequence Diagrams are a great asset for testers. They provide exactly the kind of information that testers typically never get to see, because they detail a layer that is rarely accessible by testers. Fundamentally, these diagrams provide a picture of what the testers must test and validate.

1.5 PROBLEM STATEMENT

This research work proposes generation of test cases automatically using UML models. This approach advocates a methodology that begins as soon as the software functional specification is available. This enables us to develop test cases in parallel with the software and facilitates defect detection even before code is written. Use Case Diagrams and Sequence Diagrams are used as a source of test case generation. The test suite aims to cover operational, Use Case dependency faults, various interaction and scenario faults.

A Sequence Diagram represents various interactions possible among different objects during an operation. Faults such as incorrect response to a message, correct message passed to a wrong object, message invocation with improper or incorrect arguments, message passes to yet to be instantiated objects, incorrect or missing output may occur during an interaction. These are known as Interaction faults.

Also a Sequence Diagram may depict several operation scenarios. Each scenario corresponds to a different sequence of message path in the Sequence Diagram. For a given operation scenario, the sequence of message may not follow the desired path due to incorrect condition evaluation, abnormal termination etc. These are termed as Scenario faults.

1.6 CURRENT STATUS OF THE PROBLEM TAKEN UP

Several research attempts have been reported on scenario coverage based system testing These attempts are basically black box approaches and do not take into consideration the structural and behavioral design into consideration. For testing different aspects of object interaction, several researchers have proposed different technique based on UML interaction diagrams (sequence and collaboration diagram) Bertolino and Basanieri [18] 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 and generate test cases manually following the sequences of messages between components over the Sequence Diagram. In another interesting work, Basanieri et al. [16] 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 for generating tests manually. The approach proposed in [17] generates test cases based on UML Sequence Diagram that are reverse engineered from the code under test.

1.7 RELEVANCE AND IMPORTANCE OF THE TOPIC

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 breadth-first search algorithm to generate test cases. The approach does not require any modification in the UML models or manual intervention to set input/output etc. to compute test cases. Hence, it provides a tool that straightway can be used to automate a part of testing process. A graph based methodology is followed and run-time complexity to enumerate all paths is $O(n^2)$ in the worst case for a graph of n nodes. This implies that the approach can handle a large design efficiently.

CHAPTER 2

LITERATURE SURVEY

This chapter gives a brief overview of the work that has been done by researchers in generating automatic test cases.

Automatic Test Case Generation from UML Sequence Diagrams / UML Models [1] & [2]

The authors have proposed a novel approach of generating test cases from UML design diagrams. They have used Use Case and Sequence Diagrams for the test generation scheme. Their approach consists of transforming the UML diagrams into XML code and then generating Use Case Diagram Graph (UDG) and Sequence Diagram Graph (SDG). The Graphs are then integrated to form System Testing Graph (STG). The STG is then traversed to generate the test cases.

A Survey on Automatic Test Case Generation [3]

A variety of approaches have been proposed for software testing and there has been a constant research on generating test cases based on specification and design models. A comparative analysis has been done and the authors have concluded that model based testing is more valuable than other testing techniques. Models are simple to modify, generate innumerable test sequences and allow the testers to get more testing accomplished in shorter time. Still research is being carried out to optimize the generation of test cases with minimum human effort.

Model Based Testing of System Requirements using UML Use Case Models [4]

This paper discusses a model based testing approach for creating test cases from UML Use Case models. The Use Case model is supported by Activity Diagram, Sequence Diagrams and Data Variation Equivalence Classes. The authors have used tool support to generate a test suite from this model that covers the alternative variations of each Use Case and the data variations used in each scenario. This approach has been developed in Siemens and used in a medical project.

Using UML for Automatic Test Generation [7] & [13]

As software systems are extremely complex, the amount of information contained in a system implementation is hard to comprehend in its entirety. Testing cannot be done without first understanding what the implementation is supposed to do; there should be a way to manage this complexity. The authors have created a suitable model of the system. The authors have presented an architecture for model-based testing using a profile of the UML. Class, Object, and State diagrams are used to define essential models and descriptions that characterize the entire range of possible behaviors are expressed in terms of the actions and events of the model. Object and State diagrams have been used to introduce test directives. Models written in this profile are compiled into a tool language: the Intermediate Format (IF). Descriptions written in IF are animated, verified, and used to generate tests.

The paper also illustrates the adopted testing tool, defines the profile for UML, explains testing directives, the basis of the compilation into IF and of the test generation process, and reports upon the problems encountered.

SeDiTeC - Testing Based on Sequence Diagrams [12]

This paper describes an approach to testing that uses UML Sequence Diagrams as test specification. The authors have presented a concept for automated testing of object-oriented applications and a tool called SeDiTeC that implements these concepts for Java applications. SeDiTeC uses UML Sequence Diagrams that are complemented by test case data sets consisting of parameters and return values for the method calls, as test specification and therefore can easily be integrated into the development process as soon as the design phase starts. SeDiTeC supports specification of several test case data sets for each Sequence Diagram as well as to combine several Sequence Diagrams to so-called combined Sequence Diagrams thus reducing the number of diagrams needed. For classes and their methods whose behavior is specified in Sequence Diagrams and the corresponding test case data sets SeDiTeC can automatically generate test stubs thus enabling testing right from the beginning of the implementation phase. Validation is not restricted to comparing the test case data sets with the observed data, but can also include validation of pre- and post conditions.

UML based Test Generation and Execution [11]

The authors have presented an overview of an ongoing research & development project at Siemens Corporate Research in which UML based models are being used to improve the testing. They have developed effective techniques and tools for generating a set of black box conformance tests that can be used to validate a component, subsystem or application under test. They have used category partition method which identifies behavioral equivalence classes within a structure of a system under test.

Regression Testing based on UML Design Models [17]

In this paper a methodology for identifying changes and test case selection based on the UML designs of the system have been presented. UML class diagram and Sequence Diagrams are used to generate an extended concurrent control flow graph (ECCFG) which is further used for regression testing. They have shown that their approach selects a precise set of test cases from an existing test suite.

A UML-Based Approach to System Testing [10]

The goal of this paper is to support the derivation of functional system test requirements, which will be transformed into test cases, test oracles, and test drivers once we have detailed design information. In this paper, the authors have described a methodology to address testability and automation issues, as the ultimate goal is to fully support system testing activities with high-capability tools.

Software Testing and the UML [14]

The possibility of using UML for software testing has been addressed in this paper. UML has received a great deal of attention (both positive and negative) from the software design and development communities. UML models are built extensively for Object oriented software systems. Class diagrams, State diagrams, Object Modeling Technique (OMT) and Unified process are used to test Object oriented systems. The focus of this paper is primarily on the behavioral diagrams. This is because most of the activities in software testing seek to discover defects that arise during the execution of a software system, and these defects are generally dynamic (behavioral) in nature. The author has provided a framework highlighting which diagrams are suitable for each phase of testing.

Automatic Test Case Generation using Unified Modeling Language (UML) State Diagrams [15]

UML is widely accepted and used by industry for modeling and design of software systems. A novel method to automatically generate test cases based on UML State models is presented. In the present approach, the control and data flow logic available in the UML State Diagram to generate test data are exploited. The state machine graph is traversed and the conditional predicates on every transition are selected. Then these conditional predicates are transformed and function minimization technique is applied to generate test cases. The present test data generation scheme is fully automatic and the generated test cases satisfy transition path coverage criteria. The generated test cases can be used to test class as well as cluster-level state-dependent behaviors.

Comparative Evaluation of Tests Generated from Different UML Diagrams [8]

This paper presents data comparing the use of State Charts and Sequence Diagrams for unit and integration software testing. A single project experiment on the fault revealing capabilities of model-based test sets was conducted. The tests are generated from UML State Charts and UML Sequence Diagrams. Their experiment has shown that the State Chart test sets did better at revealing unit level faults than the Sequence Diagram test sets, and the Sequence Diagram test sets did better at revealing integration level faults than the State Chart test sets. The State Charts also resulted in more test cases than the Sequence Diagrams. The results show that model-based testing can be used to systematically generate test data and indicates that different UML models can play different roles in testing.

Automated Test Case Generation from Dynamic Models [5]

The paper outlines how test suites with a given coverage level can be automatically generated from the State Charts. All the elements of a typical Use Case document are mapped to a UML State machine. Based on this state machine, artificial intelligence planning methods are applied to derive test suites with a given coverage. The application of the state of the art planning tool graph plan yields the different test cases as solutions to a planning problem. The test cases (sequences of messages plus test data) can be used for automated or manual software testing on system level.

UML-Based Statistical Test Case Generation [16]

This paper introduces an approach for generating system-level test cases based on Use Case models and refined by State Diagrams. It allows for the systematic transformation of a Use Case model of a software system into a usage model which describes both system behavior and usage. The method is intended for integration into an iterative software development process model. The resulting test cases are suited to be carried out in conventional ways, i.e., either manually or using test tools. The method is supported by an XML-based tool for model transformation. Furthermore, the use of UML models for purposes of generation encourages more extensive modeling. The information described in these models is of great use to requirements specification as well. The result is a threefold benefit: generating test cases with high efficiency and quality, more detailed information for requirements engineering, and – motivated by the more intense exploitation of the models – an encouragement for developers towards more in-depth modeling and precise maintenance of models. It aims at statistical (reliability) testing rather than fault detection.

CHAPTER 3

METHODOLOGY

This chapter discusses in detail the process of generating test cases automatically from the Sequence Diagrams.

3.1 DETAILS OF THE METHODOLOGY

A Sequence Diagram (SD) shows how a sequence of message is intended to service a single user input or an external event. A path is a sequence of transitions. The path begins with an externally generated event and ends with the production of a response that satisfies this event. A scenario is one path through a Sequence Diagram. A Sequence Diagram typically includes many scenarios, each of which should be tested. Each scenario corresponds to a different path through the Sequence Diagram. Each object or subsystem interface that participates in the scenario must be physically correct and must provide a correct implementation of its responsibilities. In addition, the overall design to produce the response must be correct. Interaction and Scenario faults can occur in a Sequence Diagram. Such faults will be found on any path through the Sequence Diagram. Hence all the paths have to be traced on the Sequence Diagram Graph (SDG).

Tracing the flows on the SD can be a visual mess. So the information in the SD can be transformed into a SDG, which is a highly testable model.

Drawing a SDG will probably reveal the ambiguities and omissions in the SD. The SDG is then traversed to generate the test cases.

The proposed method was applied and tested with an ATM simulation system. The ATM system simulates an automated teller machine (ATM). An ATM allows its users to perform basic banking operations like withdrawal, deposit, transfer and checking the balance, without having to go to the bank. In an ATM, the user inserts an ATM card, enters a PIN (a Personal Identification Number), selects a transaction to be performed and provides the necessary input for the transaction, e.g. amount and type of account in case of withdrawal. In response to the user's actions, the ATM reads and validates the card, and the PIN, processes the transaction, prints the receipt and ejects the card at the end of the session.

The example SD considered is associated with the Use Case PIN authentication in a usual ATM system which is shown in Fig 3.1. The system validates the ATM card to determine that the expiration date has not passed; that the user-entered PIN matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated. This leads to five possible scenarios which are given below.

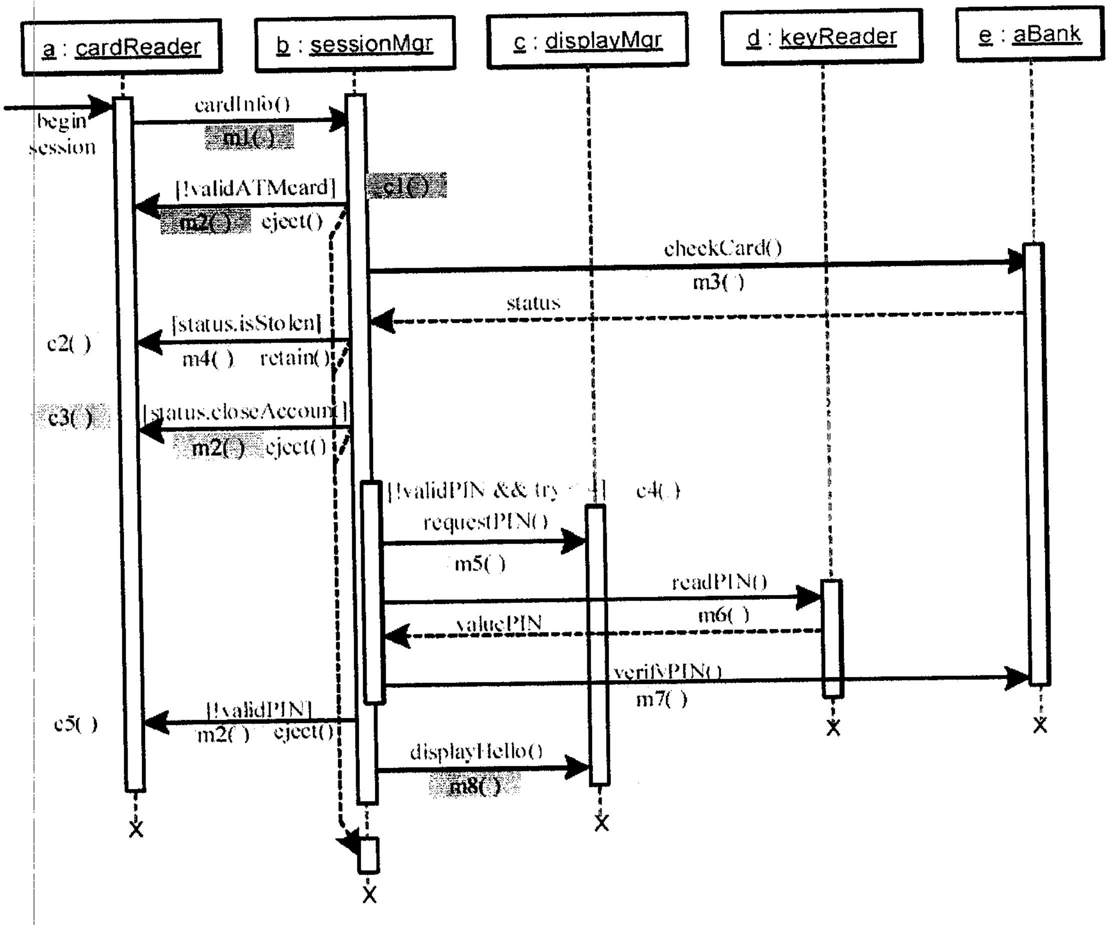


Fig. 3.1 Sequence Diagram for ATM PIN Authentication

Precondition : ATM is idle and displaying a welcome message. (for all five scenarios)

Scenario 1: ATM card is Invalid.

1. The ATM Customer actor inserts the ATM card into the Card Reader. The card input is read by the Card Reader Interface object.

2. The Card Reader Interface object sends the card input data, containing card id and expiration date to the Session Manager object.

3. The Session Manager object verifies the card input data with the Bank object and finds it is not a valid card.

4. System displays Invalid Card message.

5. System ejects card.

6. System requests the customer to take card.

7. The customer takes the card.

Postcondition : ATM is idle and displaying a welcome message.

Scenario 2: ATM card is stolen.

1. The ATM Customer actor inserts the ATM card into the Card Reader. The card input is read by the Card Reader Interface object.

2. The Card Reader Interface object sends the card input data, containing card id and expiration date to the Session Manager object.

3. The Session Manager object verifies the card input data with the Bank object and finds that the card is reported as stolen.

4. System displays Stolen Card message.

5.System displays that the card will be permanently retained by the machine.

Postcondition : ATM is idle and displaying a welcome message.

Scenario 3: ATM card whose account is closed in the bank

1. The ATM Customer actor inserts the ATM card into the Card Reader. The card input is read by the Card Reader Interface object.

2. The Card Reader Interface object sends the Card input data containing card id and expiration date to the Session Manager object.

3. The Session Manager object verifies the card input data with the Bank

object and finds that the account has been closed.

4. System displays 'Account Closed' message.

5. System ejects card.

6. System requests the customer to take card.

7. The customer takes the card.

Postcondition: ATM is idle and displaying a welcome message.

Scenario 4: Invalid PIN

1. The ATM Customer actor inserts the ATM card into the Card Reader. The card input is read by the Card Reader Interface object.

2. The Card Reader Interface object sends the Card input data, containing card id and expiration date to the Session Manager object.

3. The Session Manager object verifies the card input data with the Bank object.

4. System requests to enter PIN.

5. User enters PIN.

6. The Key reader reads the PIN.

7. The Session Manager verifies the PIN with the Bank object and finds it invalid.

8. System displays invalid PIN message.

9. System requests to enter PIN.

10. User enters PIN.

11. The Session Manager verifies the PIN with the Bank object and finds it invalid.

12. System displays invalid PIN message.

13. System requests to enter PIN.

14. User enters PIN.

15. The Session Manager verifies the PIN with the Bank object and finds it invalid.

16. System displays invalid PIN message.

17. System requests to enter PIN.

18. User enters PIN.

19. The Session Manager verifies the PIN with the Bank object and finds it invalid.

20. System displays that the card will be permanently retained by the machine.

Postcondition : ATM is idle and displaying a welcome message.

Scenario 5: Valid PIN

1. The ATM Customer actor inserts the ATM card into the Card Reader. The card input is read by the Card Reader Interface object.

2. The Card Reader Interface object sends the card input data, containing card id and expiration date to the Session Manager object.

3. The Session Manager object verifies the card input data with the Bank object.

4. System requests to enter PIN.

5. User enters PIN.

6. The Key reader reads the PIN.

7. The Session Manager verifies the PIN with the Bank object and finds it valid.

Postcondition: Display menu for transaction

Each node in the SDG stores the necessary information for test case generation. This information is collected from the Use Case template class diagrams, and data dictionary. The SDG is then traversed and test cases are generated based on coverage criteria. The block diagram and the schematic representation of this approach are shown in Fig.3.2 & 3.3 respectively.

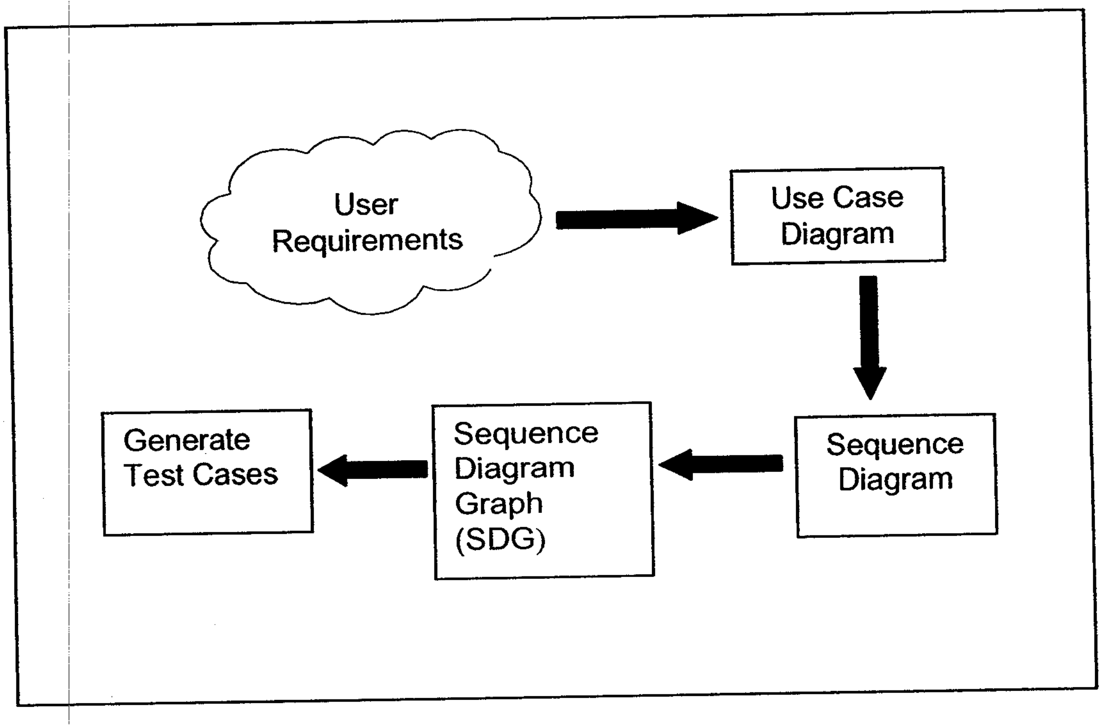


Fig. 3.2 - Block diagram

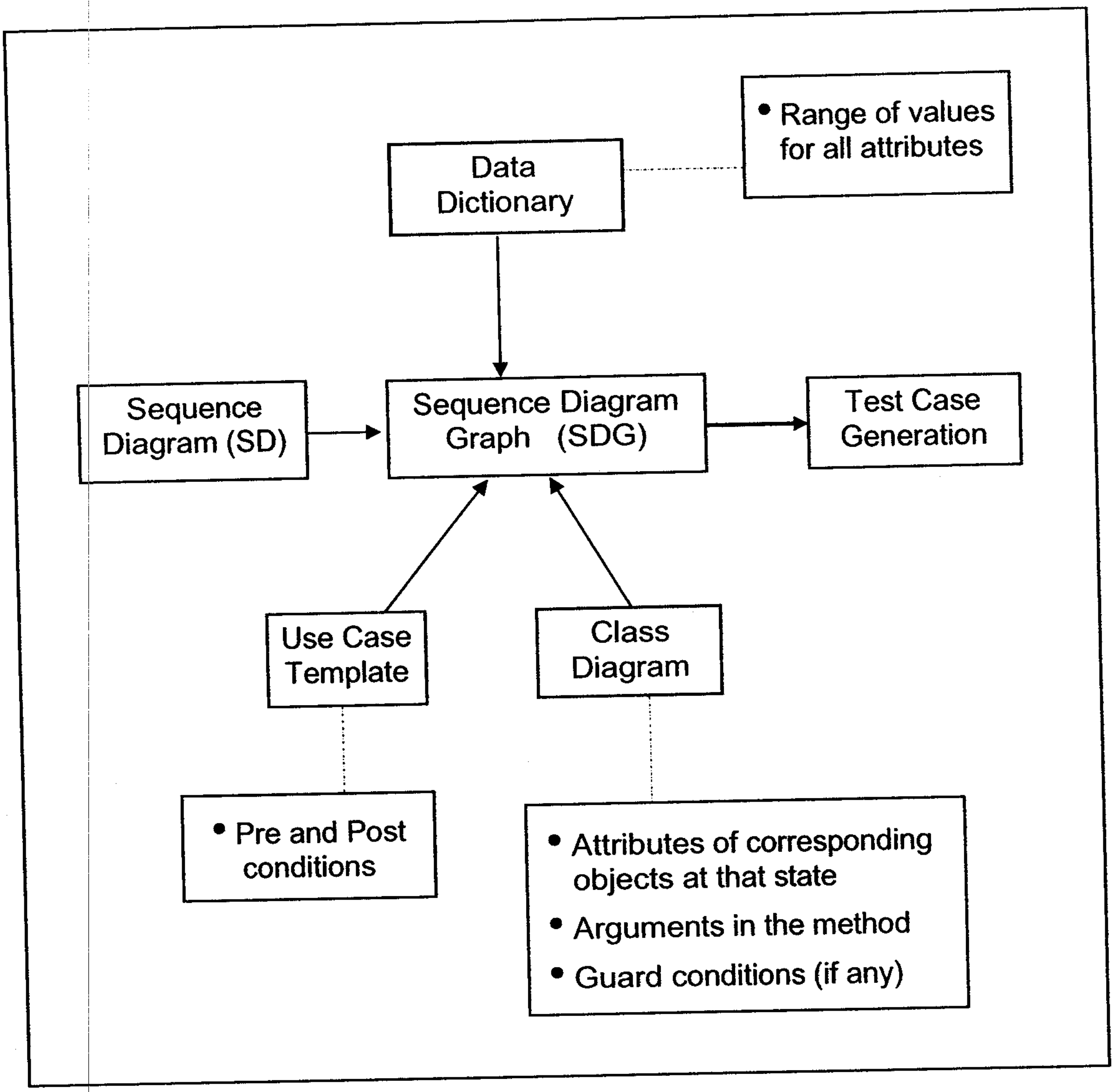

Range of values for all attributes
Data Dictionary
Sequence Diagram (SD)
Sequence Diagram Graph (SDG)
Test Case Generation
Use Case Template
Class Diagram
Pre and Post conditions
Attributes of corresponding objects at that state
Arguments in the method
Guard conditions (if any)

Fig. 3.3 - Schematic Representation

3.2 TRANSFORMATION OF AN SD INTO AN SDG

In this section, first the definition of SDG and subsequently the methodology to transform a SD into an SDG are given.

3.2.1 Definition of SDG

A Sequence Diagram Graph (SDG) is defined as

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

- $\mathbf{S_{SDG}}$ - is the set of all nodes representing various states of operation scenarios; Each node basically represents an event.
- Σ_{SDG} - is the set of edges representing transitions from one state to another.
- $\mathbf{q0_{SDG}}$ - is the initial node representing a state from which an operation begins.
- $\mathbf{F_{SDG}}$ - is the set of final nodes representing states where an operation terminates.

3.2.2 Methodology to Transform a SD Into a SDG

In order to formulate a methodology, an operation scenario is defined 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. An SDG has a single start state and one or more end states 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 the following example.

Individual aOpnScn of the Sequence Diagram (Fig 3.1) is shown in Fig. 3.4. 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. 3.4 is StateX and the two different Next States 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 scn1, there are three transitions: from StateX to s1, s1 to s2 and s2 to StateY.

3.2.3 Creation of SDG

To create the SDG for any Sequence Diagram, first identify OpnScn, the set of all operation scenarios where

OpnScn = {aOpnScn$_1$, aOpnScn$_2$, ..., }

For each aOpnScn$_i$? OpnScn, identify set of all aEvent. Initially SDG contains only the start state i.e StartState. 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. 3.1 is shown in Fig. 3.5

<scn1	<scn2	<scn3	<scn4	<scn5
StateX	StateX	StateX	StateX	StateX
s1:(m1,a,b)	s1:(m1,a,b)	s1:(m1,a,b)	s1:(m1,a,b)	s1:(m1,a,b)
s2:(m2,b,a)|c	s3:(m3,b,e)	s3:(m3,b,e)	s3:(m3,b,e)	s3:(m3,b,e)
StateY>	s4:(m4,b,a)|c	s5:(m2,b,a)|c	s6:(m5,b,c)|c4*	s6:(m5,b,c)|c4*
	StateY>	StateY>	s7:(m6,b,d)|c4*	s7:(m6,b,d)|c4*
			s8:(m7,b,e)|c4*	s8:(m7,b,e)|c4*
			s9:(m2,b,a)|c5	s10:(m8,b,c)
			StateY>	StateZ>

Fig 3.4 Five Operation Scenarios represented in the form of Quadruples

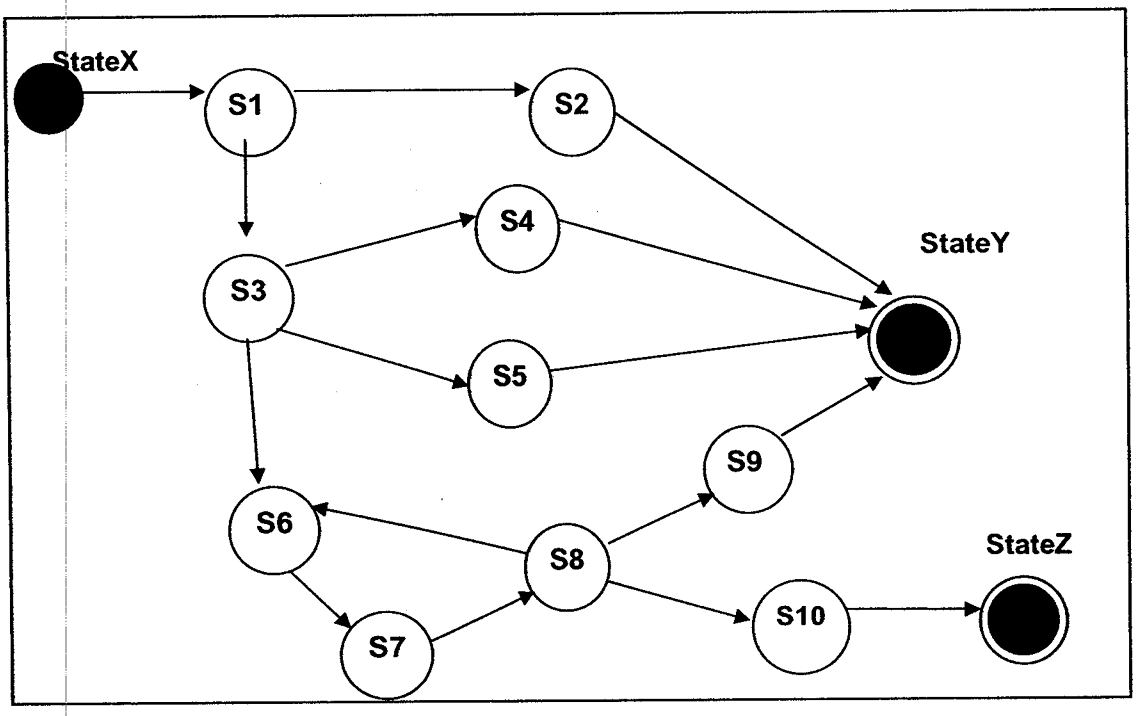


Fig 3.5 Sequence Diagram Graph (SDG)

3.2.4 Information to be stored in the SDG

SDG plays an important role in the automatic test case generation scheme. For this, SDG should contain certain necessary information for test case 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 need to be stored in its corresponding node in the SDG. The following data need 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. A guard is a predicate expression associated with an event. In addition to this, a node also stores range of values of all attributes of the objects at the state. Further, a node stores the expected results for an occurrence of an event. For example, consider a method m_i of an object o_k 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 Object Constraint Language - 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.

Once the SDG is created, it is then traversed to generate the required test cases.

CHAPTER 4

IMPLEMENTATION

This chapter explains in detail the algorithm used for generating the test cases from the SDG.

4.1 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.

The coverage criterion is "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, first all the possible paths from the start node to a final node in the SDG have to be enumerated. Each path is then visited to generate test cases. The algorithm to generate test set satisfying the coverage criterion is given below. Every test strategy targets to detect certain categories of faults called the fault model .This test strategy is based on the discovering interaction and scenario faults.

4.1.1 Algorithm *TestSetGeneration*

Input: Sequence diagram graph *SDG*

Output: Test suite *T*

```
1. Enumerate all paths P= {$P_1$, $P_2$, $P_3$,... $P_n$} from the start node to a final
   node in SDG.
2. For each path $P € P_i$ do
3. $n_x = n_j$                // $n_j$ 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. $t_i \leftarrow F$            // 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 ( $n_j <> n_z$ ) do              // $n_z$ being a final node
8.     $e_j$ = { m, a, b, c } // The event corresponding to the node $n_j$ and m(...)
                                 is invoked with a set of arguments $a_i$, $a_2$,...,$a_l$
9.     If c = null then    //If there is no guard condition
10.        Select test case t = { preC, I ($a_1$, $a_2$,....$a_i$), O( $d_1$, $d_2$,...$d_m$), postC }
               where preC = precondition of the method m
               I ($a_1$, $a_2$,....$a_i$) = set of input values for the method m(...)from
               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(...)
11.            Add t to the test set $t_i$, that is $t_i = t_i \cup t$
12.     EndIf
13.   If c < > A, then                   //method m is under guard condition
14.     c(v) = ( $c_1$, $c_2$,.... $c_i$)    // The set of value of clauses on the path Pi
```

```
15.   Select test case  t = { preC, I ($a_1$, $a_2$,....$a_i$), O($d_1$,$d_2$,...$d_m$),, c(v), postC}
                where *preC* ≐ precondition of the method *m*
                I ($a_1$, a ,....$a_i$) = set of input values for the method *m*(...)
                obtained from *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*(...)
16.        Add *t* to the test set $t_i$, that is $t_i$ = $t_i$ U t
17.   EndIf
18.        $n_j$ = $n_k$            // Move to the next node $n_k$ on the path $P_i$
19         $T \leftarrow T$ U $t_i$
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 U t
24.   EndFor
25.   Return (*T*)
26.   Stop
```

CHAPTER 5

CONCLUSION AND FUTURE WORK

An approach to automatically generate test cases from Sequence Diagrams has been implemented in C. The approach does not require any modification in the UML models; it provides a tool that can be used to automate the testing process. The traversal of SDG is done using breadth first search and the time complexity is $O(n^2)$ in the worst case for a graph of 'n' nodes.

The work could be extended further by using different traversal algorithms for generating the test cases and doing a comparative analysis on the time complexity. Also, methods could be devised for detecting faults other than Interaction and Scenario faults.

APPENDIX

SAMPLE INPUT

```
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Fig A.1 Adjacency Matrix Representation of the SDG

Precondition: ATM is idle and displaying a welcome message. User inserts a card.+0

Read card info.+0

Card = "Not ATM"+0

Card = "ATM"+0

Status = "Stolen"+0

Account = "Closed"+0

Status = "Okay" Account = "Open"+0

Enter PIN+0

PIN = "Invalid" Message "Invalid PIN: Try Again"| PIN = "Valid"+1+3+6

"Try Later"+0

Display "Hello"+0

Eject card. Postcondition: Display Welcome Message+0

Eject card.+0

Fig A.2 Message Inputs for the SDG

SAMPLE OUTPUT

Test case Scenario 1

Precondition : ATM is idle and displaying a welcome message. User inserts a card.
Read card info.
Card = "Not ATM"
Eject card.
Postcondition : Display Welcome Message

Test case Scenario 2

Precondition : ATM is idle and displaying a welcome message. User inserts a card.
Read card info.
Card = "ATM"
Status = "Stolen"
Eject card.
Postcondition: Display Welcome Message

Test case Scenario 3

Precondition: ATM is idle and displaying a welcome message. User inserts a card.
Read card info.
Card = "ATM"
Account = "Closed"
Eject card.
Postcondition: Display Welcome Message

Test case Scenario 4

Precondition : ATM is idle and displaying a welcome message. User inserts a card.

Read card info.

Card = "ATM"

Status = "Okay" Account = "Open"

Enter PIN

PIN = "Invalid" Message "Invalid PIN : Try Again"

PIN = "Valid" Status = "Okay" Account = "Open"

Enter PIN

PIN = "Invalid" Message "Invalid PIN : Try Again"

PIN = "Valid" Status = "Okay" Account = "Open"

Enter PIN

PIN = "Invalid" Message "Invalid PIN : Try Again"

PIN = "Valid" Status = "Okay" Account = "Open"

Enter PIN

Try : 0 "Try Later"

Eject card.

Postcondition: Display Welcome Message

Test case Scenario 5

Precondition: ATM is idle and displaying a welcome message. User inserts a card.

Read card info.

Card = "ATM"

Status = "Okay" Account = "Open"

Enter PIN

PIN = "Valid" Display "Hello"

Postcondition: Display Menu for Transaction

REFERENCES

[1] Monalisa Sarma, Debasish Kundu, Rajib Mall, "Automatic Test Case Generation from UML Sequence Diagrams", International Conference on Advanced Computing and Communications, 2007

[2] Monalisa Sarma, Rajib Mall, "Automatic Test Case Generation from UML Models", International Conference on Information Technology, 2007

[3] M Prasanna, S N Sivanandam, R Venkatesan, R Sundarrajan, "A Survey on Automatic Test Case Generation", ACAD Journal, 2005

[4] Bill Hasling, Helmut Goetz, Klaus Beetz, "Model Based Testing of System Requirements using UML Use Case Models", International Conference on Software Testing, Verification and Validation, 2008

[5] Peter Frohlich, Johannes Link, "Automated Test Case Generation from Dynamic Models", LNCS 1850, Springer, 2000

[6] Noraida Ismail, Rosziati Ibrahim, Noraini Ibrahim, "Automatic Generation of Test Case from Use Case Diagram", International Conference on Electrical Engineering and Informatics, 2007

[7] Alessandra Cavarra, Jim Davies, Thierry Jeron, Laurent Mounier, Alan Hartman, Sergey Olvovsky "Using UML for Automatic Test Generation", Automated Software Engineering Conference, 2001

[8] Supaporn Kansomkeat, Jeff Offutt, Aynur Abdurazik, Andrea Baldini, "A Comparative Evaluation of Tests Generated from Different UML Diagrams", International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2008

[9] Dehla Sokenou, "Generating Test Sequences from UML Sequence Diagrams and State Diagrams", GI Jahrestagung, 2006

[10] Lionel Briand, Yvan Labiche, "A UML-Based Approach to System Testing", International Conference on the Unified Modeling Language, Modeling languages, Concepts and Tools, 2001

[11] Jean Hartmann, Marlon Vieira, Herb Foster, Axel Ruder, "UML-based Test Generation and Execution", International Conference on the Unified Modeling Language, Modeling languages, Concepts and Tools, 2001

[12] Falk Fraikin, Thomas Leonhardt, "SeDiTeC - Testing Based on Sequence Diagrams", International conference on Automated Software Engineering, 2002

[13] Charles Crichton, Alessandra Cavarra, and Jim Davies, "Using UML for Automatic Test Generation", Automated Software Engineering Conference, 2001

[14] Clay E. Williams, "Software Testing and the UML", International Symposium on Software Reliability Engineering, 1999

[15] P.Samuel, R. Mall, A.K.Bothra, "Automatic Test Case Generation using Unified Modeling Language (UML) State Diagrams", IET Software, 2008

[16] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow Suite Approach to Planning and deriving Test Suites in UML projects", International Conference on the UML, LNCS, 2002

[17] P. Tonella, and Potrich, A. "Reverse Engineering of the Interaction Diagrams from C++ Code", IEEE International Conference on Software Maintenance, 2003

[18] A.Bertolino, and F.Basanieri, "A Practical Approach to UML-based Derivation of Integration Tests", International Software Quality Week Europe, Brussels, Belgium, 2000.