B

UML BASED REGRESSION TESTING TECHNIQUE FOR OO SOFTWARE

Rt
479
c.1

by

HUSAM TAKKOUSH

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

103374

Thesis Advisor: Dr. NASHAT MANSOUR

Department of Computer Science

LEBANESE AMERICAN UNIVERSITY

February 2006

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences - Beirut Campus

Thesis approval Form (Annex III)

Student Name: Hussam Takrouch I.D. #: 199910050

Thesis Title : UML Based Regression Testing Technique for OO software

Program : MS Computer Science

Division/Dept : Div. of Comp. Sc and Maths.

School : **School of Arts and Sciences**

Approved by:

Thesis Advisor: Nashat Mansour

Member : HAIDAR HARMANANI

Member : Faisal Abu Khzam

Date 16/2/06

I grant the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the university or to its students, agents and employees. I further agree that the university may reproduce and provide single copies of the work, in ay format other than in or from microforms, to the public for the cost reproduction.

____

ACKNOWLEDGMENTS

I would like to express my gratitude to my teacher and thesis supervisor Dr. Nashat Mansour for all the help, guidance, and support he gave me throughout the work. I would like to thank him especially for always answering my questions promptly and for reviewing and commenting on many drafts and papers.

I would also like to thank the committee members, Dr. Faisal Abu Khzam and Dr. Haidar Harmanani for their valuable advises.

A special note of thanks goes to my parents and friends, for their help and encouragement.

Finally, I would like to thank my wife, Imène, for her patience during the past few months while I carried out this work.

To my parents and wife

UML BASED REGRESSION TESTING TECHNIQUE FOR OO SOFTWARE

ABSTRACT

by

HUSAM TAKKOUSH

In this thesis we present a technique for regression testing of object oriented software based on the unified modeling language (UML). UML is a widely accepted modeling language for object oriented software. Regression test selection is important because it saves both time and cost by reducing the number of test cases to be performed for validation. The technique presented in this thesis selects test cases from a pool of unit test cases as well as another pool of integration test cases. This test case selection technique is based on the design without access to source code. The UML design diagrams give us more insight of the interactions and dependencies than black box testing. Our approach utilizes two UML diagrams, the class and interaction overview. Class diagrams provide us with the blue print of the object oriented software including all the classes and methods. Interaction overview diagrams were newly introduced in UML2.0; however, their nature as a combination of interaction diagrams and activity diagrams makes them very useful in process modeling and test case selection.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Regression Testing 1
1.2 UML 2
1.3 Design Based Testing and Regression Testing 3
1.4 Objectives and Scope of the Work 3
1.5 Organization of the Thesis 4

CHAPTER 2 RELATED WORK 5
2.1 Firewall Approach 5
2.2 Using Control Flow Graphs 5
2.3 Using Function Dependency Graphs 6
2.4 Other Classical Techniques 6
2.4.1 Simulated annealing 6
2.4.2 Reduction 7
2.4.3 Slicing 7
2.4.4 Dataflow 7
2.5 Using UML 7
2.5.1 Class and Sequence Diagrams 8
2.5.2 Collaboration and State Chart Diagrams 8

CHAPTER 3 CHANGES IN OO DESIGN 10
3.1 Changes in Class Diagrams 10
3.1.1 Attributes Changes 12
3.1.2 Method Changes 13
3.1.3 Relationship Changes 13
3.1.4 Class Changes 14
3.2 Changes in Interaction Overview Diagrams 14
3.2.1 Activity: Addition/Update/Delete 17
3.2.2 Flow/edge: Addition/Update/Delete 17
3.2.3 Decision: Addition/ Delete 18
3.3.4 Flow final: Addition/Delete 18
3.3 Changes inside Sequence Diagrams 20
3.3.1 Lifelines Changes 20
3.3.2 Messages Changes 21

CHAPTER 4 TEST SELECTION FOR REGRESSION TESTING 23
4.1 Notation 24
4.2 Assumptions 25
4.3 Description of Technique 26
4.4 Test Case Selection Algorithm 28
4.4.1 Test Selection Based on Class Diagrams 28
4.4.1.1 Directly Changed Methods in a Class Diagram 29
4.4.1.2 Integration Test Case Selection from Method Changes 30
4.4.1.3 Unit Test Case Selection from Method Changes 31
4.4.2 Test Selection Based on Interaction Overview Diagrams 33
4.4.2.1 Interaction Overview Diagram Based Classification 35
4.4.2.2 Sequence Diagrams Based Reduction 37

CHAPTER 5 CASE STUDY 39
5.1 Changes Tracking 39

5.1.1 Changes in the Class Diagram 39
5.1.2 Changes in the Sequence Diagrams 41
5.1.3 Changes in Interaction Overview Diagrams 42
5.2 Original Suit of Test Cases: 42
5.2.1 Integration Test Cases Path in the Interaction Overview Diagram. 43
5.2.2 Integration Test Cases Path in the Sequence Diagrams. 44
5.2.2 Integration Unit Test Cases. 46
5.3 Test Case Selection 47
5.3.1 Test Case Selection Based on Class Diagram 47
5.3.2 Test Case Selection Based on Changes in the Interaction Overview Diagram 47

CHAPTER 6 CONCLUSION AND FUTURE WORK 49

BIBLIOGRAPHY 50

LIST OF FIGURES

Figure 1 ATM class diagram 11
Figure 2 ATM partial class diagram after update 12
Figure 3 ATM Interaction Overview before update 16
Figure 4 ATM Interaction Overview diagram after update 19
Figure 5 ATM Cash Withdrawal SD 22
Figure 6 Class Diagram Test Case Selection Algorithm 29
Figure 7 Generate Directly Changed Methods Algorithm 30
Figure 8 Class Diagram Integration Test Selection 31
Figure 9 Class Diagram Integration Test Selection 32
Figure 10 System Test Case Selection Algorithm 34
Figure 11 IO Based Classification/Selection Algorithm 36
Figure 12 SD Based Classification/Selection Algorithm 38
Figure 13 SD Reservation of Money 41
Figure 14 SD Reservation on Money after update 42

LIST OF TABLES

Table 1 Notations used in the algorithm........24
Table 2 ATM example integration test cases path in the interaction overview diagram (T.IO)........43
Table 3 ATM example integration test cases path in the sequence diagrams (T.SD........44
Table 4 Unit Test Cases (UT)........46

Chapter 1

Introduction

In a software lifecycle, testing software in the maintenance phase is very important. However, testing the entire system would not be favorable due to cost and time issues. Regression testing approaches aim to reduce the number of test cases to be performed on the system. In the majority of the object oriented (OO) regression testing approaches and techniques, there seem to be lack of standardization in the use of diagrams ranging from the use of Control Flow Diagrams, Interprocedural Control Flow Diagrams, Class Control Flow Diagrams, Class Dependency Relation Diagrams, to Object Relational Diagrams, and so on.

UML, on the other hand is a standard modeling language for OO programs, it represents a note to one relationship between the actual classes and the design. Our approach is to make use of two of these diagrams, the class and the interaction overview diagrams to select two different types of test cases from the original test suite: the unit tests and the integration tests.

1.1 Regression Testing

In software maintenance, testing the entire system after modification is not feasible. "To save effort and time, regression testing need only retest those parts that are affected by the modification" (Kung et al., 1996). Regression testing techniques aim to generate a reduced set of test cases selected for retesting the modified software (Mansour and Bahsoon, 2002).

Due to time constraints as well as the cost of repeating all the tests, changes in the system may require a set of test cases to be selected from the original test suite to be repeated, this is known as the regression test selection problem.

In this thesis, we introduce a new, safe, regression test selection algorithm that selects and reduces the selected test cases by utilizing different UML design diagrams. UML 2.0 has some new introduced diagrams which we utilize to gain as much information about the system in order to gain precision.

1.2 UML

The UML is a widely accepted standard for object oriented software modeling. There are 13 official diagram types in UML2.0 which are fully capable of describing the programs at all their phases, from design to implementation and execution. These diagrams are not programming language dependent. A design in UML can be implemented in Java, C#, C++, Delphi, or any other OO programming language.

In our approach, we utilize class and interaction overview diagrams in our regression test selection technique. Class diagrams are the blue print of the entire system; they represent classes and their relational dependencies. "A class diagram describes the types of objects in the system and various kinds of static relationships that exist among them" (Fowler, 2003). Interaction overview diagrams are newly introduced in UML2.0; they are a combination of activity diagrams and interaction diagrams. "Interaction overview diagrams are activity diagrams that show interactions and interaction occurrences" (Arlow and Neustadt, 2005). In our approach we utilize interaction overview diagrams that have sequence diagrams for their activity diagrams. Although new in UML 2.0, interaction overview diagrams provide a clearer, modular, view of the procedural logic.

1.3 Design Based Testing and Regression Testing

When working with large and complex systems, source code-based testing and regression testing is not feasible. "The amount of information contained in an implementation is hard to comprehend in its entirety" (Gavarra et al., 2003). Furthermore, the source code may not be accessible to testers. Testers need only to have insight of the specs to produce test cases.

Another issue with large systems is that source code is not updated by one person and most probably, with a system change, more than one class and relationship will be affected. To keep track of all these changes, changes should be documented and the design updated. Basically whatever changes in the code, the UML design should be able to reflect it. Today's CASE tools are advanced enough to detect changes in the source and update the UML design to reflect it and vice versa.

1.4 Objectives and Scope of the Work

The objective of this thesis is to present an algorithm for regression test selection based on recent standard UML diagrams. We use the newly introduced interaction overview diagram in UML 2.0 because it provides a modular overview of the system including control flow, action states, and interaction diagrams. We also employ information from class and sequence diagrams to refine our test selection. We have two test case pools for the unit and integration test cases. Based on the software changes reflected in the class and the interaction overview diagrams, the technique for test case selection is mainly composed of two cases:

In the first case we detect the changed methods in the class diagram; this is done under the assumption that the developers update the design properly. We select both unit and test cases for retest that directly traverse the changed methods. If a method is changed, those that have dependency on that method need to be retested as well. We utilize the interaction overview diagram's sequence diagrams to detect dependencies among methods.

In the second case, we detect integration level changes in the interaction overview diagram. If the change is on the action level, which is basically a sequence diagram, only the test cases with changed method sequence will be selected.

1.5 Organization of the Thesis

The remainder of this thesis is divided into 5 chapters. Chapter 2 describes pervious work related to testing and regression testing for OO software as well as some recent approaches in UML based testing and regression testing. Chapter 3 describes in detail the possible changes in the class, interaction overview, and sequence diagrams. Chapter 4 describes the algorithm of the approach used for test selection for regression testing. Chapter 5 presents an example case study of this approach. Finally, a conclusion is drawn in Chapter 6

Chapter 2

Related Work

In this chapter, we introduces the major previous work in regression testing techniques for solving the test selection problem that are relevant to OO programming, or have been improved to use OO software.

2.1 Firewall Approach

White and Leung presented this in 1993. The concept is based on creating a segment firewall in the CFG so that the system can be decomposed and the effect of a change can be traced within the firewall, and all test cases that use code in the segment firewall are selected for re-testing.

Hsia et al. (1997) also use the firewall approach in OO class testing, "selecting relevant test cases and test strategy". The Relations between classes are used to create the firewalls. "The class firewall for a class C is defined as a set of classes that are dependent on C as described by an ORD". The approach adopted by Hsia et al (1997) was to determine the relations between classes and the test cases. Then select test cases that traverse the changed classes.

2.2 Using Control Flow Graphs

Rothermel et al. (2000) moves from using CFG (Control Flow Graphs) to ICFG (Interprocedural Control Flow Graphs) because CFGs are used to cover only the inner works of each procedure while ICFG can show the system with all the procedure calls. In ICFGs, calling procedures is done by using *call* and *return* nodes,

and for each test case there will be an edge trace that will trace the execution of the code. "The goal is to identify test cases in T that execute changed code with respect to P and P'." Rothermel et al. (2000).

However, the ICFGs couldn't deal with polymorphism and other OO features, so ICFG has been extended to CCFG (Class Control Flow Graph) by the use of frames. Frames will handle the OO features like polymorphism, calling other classes and so on. With a CCFG, OO programs can be represented and the algorithm used in ICFGs can be used on CCFGs in order to avoid the complexity of polymorphism.

2.3 Using Function Dependency Graphs

Wu et al. (1999) generated an "Affected Function Dependency Graph (AFDG)", based on the functions and their effects. They classified the effects on the functions to two types: "Behavior Affected" and "non-Behavior affected".

a. "By noticing that all of these OO features are related to the function calls which are associated with certain objects, we propose a regression testing technique based on the analysis of the dependence relationships among functions in the system" (Wu et al., 1999).

2.4 Other Classical Techniques

Various classical approaches for regression testing were discussed in Mansour et al. (2001). From these approaches, we mention the simulated annealing, reeducation, slicing, dataflow, and segment-firewall.

2.4.1 Simulated annealing

Suggested in Mansour and El-Fakih(1999): it uses simulated annealing to solve the regression test selection problem. Annealing is achieved by starting at an initial high temperature and reducing the temperature gradually to freezing point

2.4.2 Reduction

Reduction was proposed in 1993 by Harrold et al.: it is related to reducing the size of the test suite to cover all the requirements with minimum number of test cases selected. This is very important for selecting the appropriate test cases to run so full requirement coverage is achieved with minimal run of tests.

2.4.3 Slicing

Agrwal et al. presented this algorithm in 1993. Slicing is based on taking a dynamic slice of the system when it is running test cases and matching if the changed code is accessed in that slice. It is based on statement coverage for a slice of a test case.

2.4.4 Dataflow

Gupta et al. presented this algorithm in 1996 and it is based on dataflow and uses backward and forward walk procedures to select the slices.

2.5 Using UML

UML has become the standard in OO design. Recent regression testing techniques adopt UML diagrams as the standard diagrams for modeling. We will introduce two adopted approaches, one that makes use of class and sequence diagrams, while the other makes use of collaboration and state chart diagrams.

2.5.1 Class and Sequence Diagrams

Briand et al. (2002) use UML to design and classify the test cases into: Retestable, Reusable, and Obsolete. Their approach is to consider changes by comparing class and sequence diagrams. After that comparison, use cases having changed sequence diagrams were classified. Test cases related to these use cases were classified as well. It is based on information from class diagrams and sequence diagrams. Basically, their algorithm goes as follows:

i. Compare the two class diagrams

ii. Compare the sequence diagrams for each test case

iii. Classify the test cases

  a. Obsolete: if it "consists of an invalid execution sequence of boundary class methods".

  b. Retestable: Test cases that need to be executed again due to change in the methods or the method order.

  c. Reusable: Test cases that need not to be run because no change impacted their use.

2.5.2 Collaboration and State Chart Diagrams.

Wu et al. (2003) used collaborative diagrams / sequence diagrams to represent interactions among different objects in the component. They also used state chart diagrams to characterize internal behaviors of objects in a component. Assuming one interface only includes one operation (Ref = Operation), they used the notions of context dependency and content dependency among components. Context dependent: "An event e2 has a context-sensitive dependence relationship with event el if there exists an execution path where triggering of el -directly or indirectly- triggers e2"

(Wu et al., 2003). Content dependent: "An interface v2 has a content-dependence relationship on interface v1 if and only if v1 contains the signature of f1, v2 contains the signature of f2 and f2 depends on f1" (Wu et al., 2003). Since Wu et al. (2003) was about component testing using UML. Their testing approach was to cover all content and context dependencies, as well as all the transitions.

Chapter 3

Changes in OO Design

Assuming that the developers have maintained a proper correlation between changes in the software source code and the UML design, our regression test selection algorithm is solely based on UML diagrams. The design diagrams used in our test case selection algorithm are class diagrams and interaction overview diagrams. Of course, interaction overview diagrams contain other types of activity diagrams. Our focus is on interaction overview diagrams assuming all actions are modeled as sequence diagrams.

In this section, we classify the changes in the class diagrams, interaction overview diagrams, and sequence diagrams.

3.1 Changes in Class Diagrams

Class diagrams: Class diagrams are blue print of the entire software. Whenever there is an implementation of a change in the source of the system, it should be clearly reflected in the class diagram as we show in this section.

Consider the class diagrams of the system, the original class diagram (CD), see Figure 1, and the new class diagram (CD'). The expected changes in the class diagram could affect any of its entities, the main entities are:

- Attributes
- Methods
- Relationships
- Classes

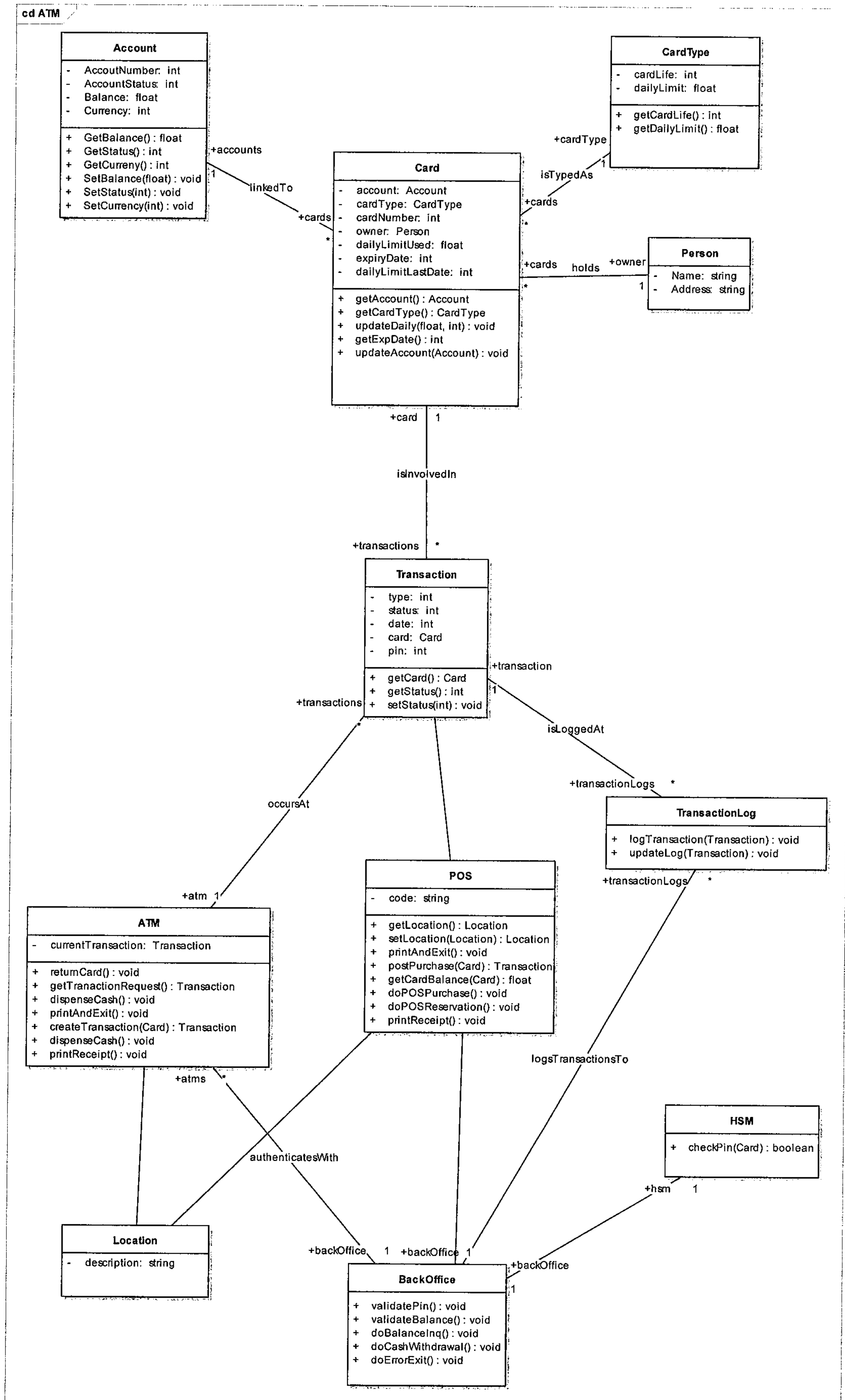

Figure 1 ATM class diagram

3.1.1 Attributes Changes

Attributes are specific to a class and changes on attributes are of three types:

- New attribute: new attributes can be added to a class in the class diagram. For example, CardType.montlyLimit, Card.monthlyLimitUsed, etc… (see Figure 2)
- Deleted attribute: an attribute may be deleted after all references to it have been removed.
- Modified attribute: an attribute could have a change in: visibility, type, multiplicity, or default value. Renaming in attributes is not a standard procedure considering dependencies and overloading. A renamed attribute is classified as a deletion of the old attribute and an addition of a new attribute with the new name.

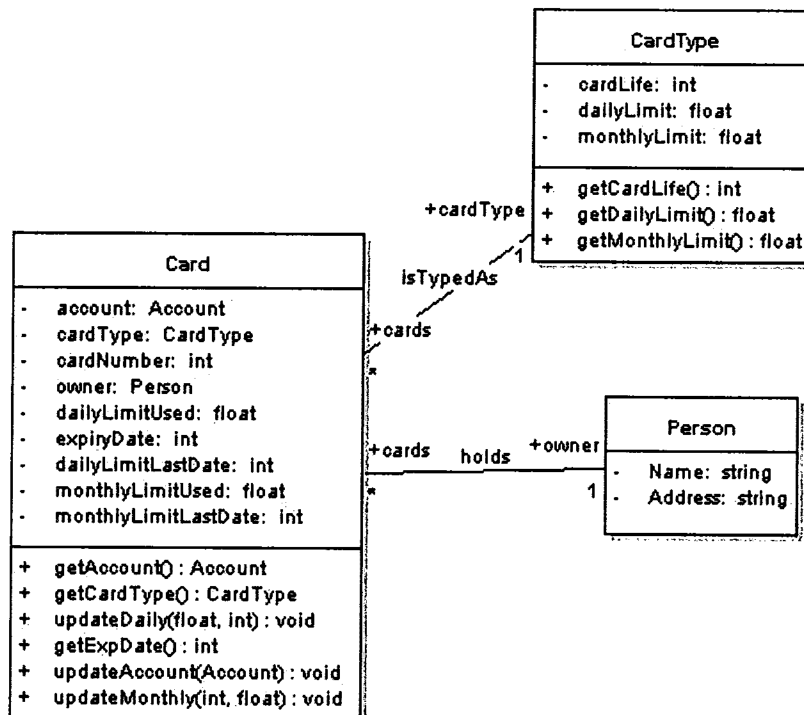

Figure 2 ATM partial class diagram after update

3.1.2 Method Changes

Method changes are similar to the attribute changes; their types are also classified into three changes, New, Deleted, and Modified. Provided that the method is identified by its signature; a change in signature can be assumed to be a deletion and addition of a new method (Briand et al., 2002)

- New method: if a new method signature is introduced in CD'. For example, Card.updateMonthly, CardType.getMonthlyLimit.
- Deleted method if a method signature is changed or deleted.
- Modified method: Methods are classified as modified if a modification occurs in:
  - Method implementation without changing the contract (due to refactoring, code optimization, etc…).
  - Change in Contract (post-conditions, pre-conditions, and invariants [Meyer]) is a change in implementation. For example, if we consider the conditions of BackOffice.validateLimit, after adding the monthly limit checking, that method contract have been changed, hence it is marked as Changed.
  - One of the attributes used in the contract belongs to the set of changed Attributes.

3.1.3 Relationship Changes

Changes between relationships are reflected in the classes by changes in the methods and attributes. Basically the changes in relationships are:

- New relationship

- Deleted relationship
- Modified relationship: modifications in relationships are modifications in the relationship type, multiplicity, and directions.

3.1.4 Class Changes

- New class: if a new class is introduced to the system.
- Deleted class: if a class was renamed or deleted.
- Modified class: A class is said to be changed if there is changes in its attributes, methods. For example, Classes that are modified in the ATM are: CardType, Cards, and BackOffice.

3.2 Changes in Interaction Overview Diagrams

Interaction overview diagrams are types of UML activity diagrams which overview the control flow of the entire system. As interaction overview diagrams are combinations of interaction diagrams and activity diagrams.

One way to look at the interaction overview diagram would be the same as activity diagrams after replacing the nodes with Frames. These Frames are actually interaction diagrams that perform a set of tasks. Basically the interaction overview diagram consists of the same components as the interaction diagram:

- Initial node.
- Activity final node.
- Activity.
- Flow/edge.
- Fork
- Join.

- Condition.
- Decision.
- Merge.
- Partition.
- Sub-activity indicator.
- Flow final.
- Note.
- Use case.

Any update to the system under test is first reflected in the class diagram, and in the interaction overview diagrams for behavioral changes.

Updates in the interaction overview diagram can be divided into two classes: changes in the components (e.g. initial node, activity, edges, etc...), or changes inside the components themselves. Basically the two major changes in the interaction overview diagrams are:

- Interactions between sequence diagrams.
- Sequence diagrams changes.

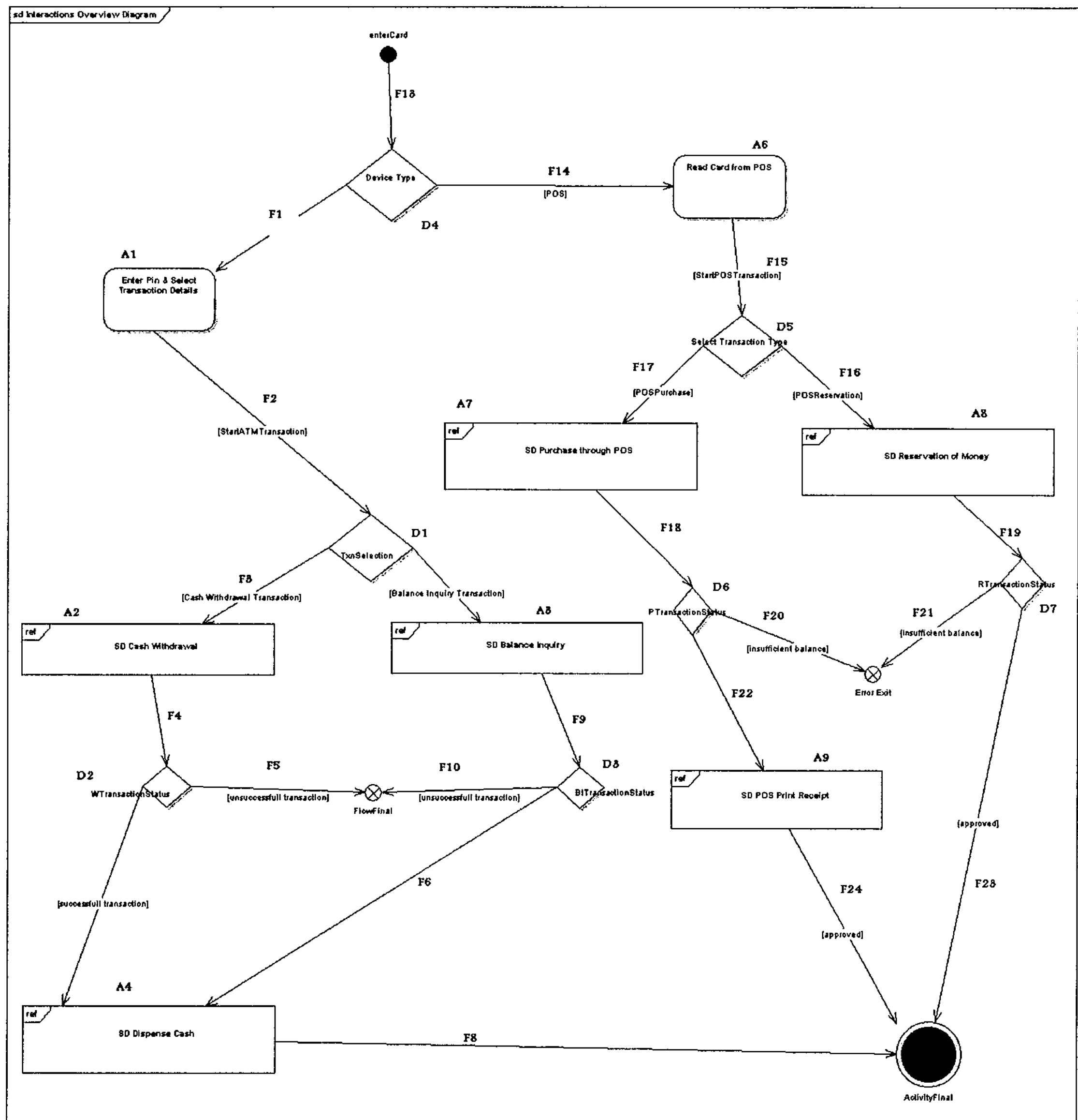

Figure 3 ATM Interaction Overview before update

Interaction overview diagrams changes that are relevant to our testing are changes in one of their components and are classified as: activity, flow/edge, decision, and flow final.

3.2.1 Activity: Addition/Update/Delete

- Changes in the activity diagrams are either an Addition of a new Activity or an update inside this activity diagram which will be further discussed in section 3.3.

For example, consider Figure 4, the SD Cash Withdrawal was updated to handle monthly limit checking (That was reflected by the change in the method validateLimit that is referenced in SD Cash Withdrawal). Also, in the same example, a new activity [printBalanceAndLimit] has been added to the flow.

3.2.2 Flow/edge: Addition/Update/Delete

- Flow/edge resemble flow of control, it can be updated by changing its condition, or source and/or target. If a change was recorded in both condition and one of the targets, then that is not an update to the relevant edge, but rather a deletion of the old one and creation of a new one.

For example, a change was recorded in the [successful transaction] edge in the balance inquiry branch to handle printing of a special receipt different than the original one, which was common with Cash Withdrawal.

3.2.3 Decision: Addition/ Delete

- The decisions nodes themselves can be either added or deleted.
- There are no updates in the decision nodes because that will involve a change in decision rendering the old decision as deleted and the update as a newly introduced decision.

3.3.4 Flow final: Addition/Delete

- Flow final nodes resemble the termination of the process. They can be either added or deleted. Unconnected flow final nodes will be deleted.

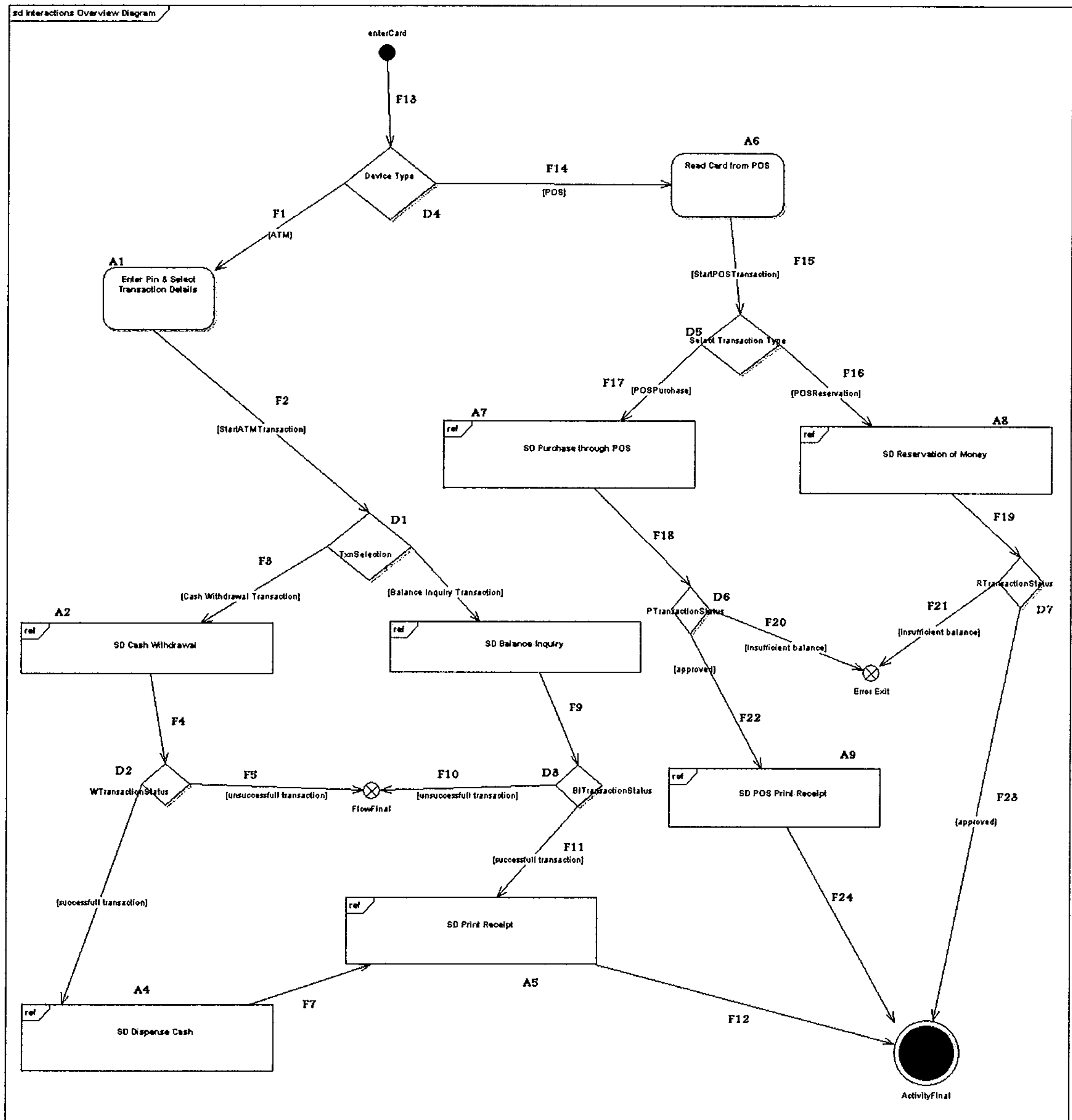

Figure 4 ATM Interaction Overview diagram after update

3.3 Changes inside Sequence Diagrams

"Sequence diagrams show interaction between lifelines as a time-ordered sequence of events. They are the richest, and most flexible, form of interaction diagram." (Arlow and Neustadt, 2005) Let SD and SD' be the original and the new sequence diagrams frames of the system under test respectively. The expected changes in the sequence diagram could affect any of its entities, the two main entities (that concern us for our regression test selection technique) are:

- Lifelines
- Messages

3.3.1 Lifelines Changes

As the lifeline represents participants in the system, the lifeline can be an object, an actor, or any control or entity element...

The expected changes in the lifelines of a sequence diagram are:

- New lifeline: new lifelines can be introduced to the sequence diagram.
- Deleted lifeline: a lifeline with no messages heading in or out from it can be deleted from the sequence diagram.
- Modified lifeline: a modification in the lifeline itself would be a modification in the sequence of execution occurrences on the same lifeline. Other types of execution occurrences may include loops, invariants, alternatives, guards, etc...

3.3.2 Messages Changes

Messages are invokers of methods. In a sequence diagram, the possible changes to the messages in the sequence diagram would be:

- Deleted message: if the method referenced by that message is deleted, or renamed.
- Modified message: a message is said to be modified if the method it references is marked as a modified method in the class diagram after the differences of class diagrams have been processed for classification.

In the ATM example, Figure 6, as the BackOffice.validateLimit() has been updated. Hence, in the SD Cash Withdrawal, the self-call message from BackOffice, BackOffice.validateLimit is marked as modified.

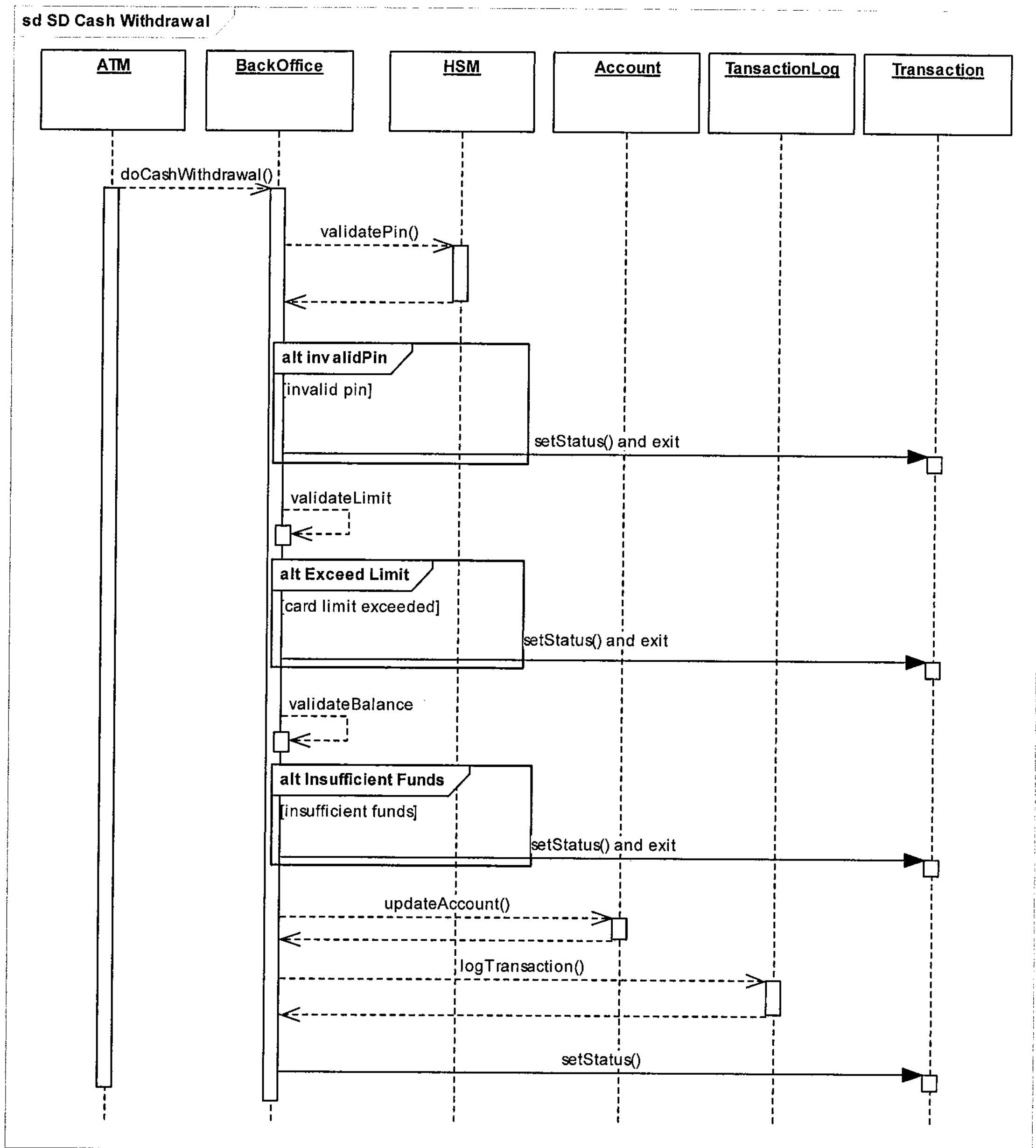

Figure 5 ATM Cash Withdrawal SD

Chapter 4

Test Selection for Regression Testing

The regression test selection algorithm is mainly based on interaction overview diagrams, which summarize the control flow of the entire system. Class diagram changes are also used in our algorithm because changes in the methods will be reflected in the interaction overview diagrams, specifically in the sequence diagrams. In this chapter, we present the algorithm for regression test case selection of both integration and unit test cases based on information derived from class and interaction overview diagram.

4.1 Notation

Table 1, contains a list of notation used in the rest of this chapter.

Table 1 Notations used in the algorithm

Notation	Description
CD	Class diagram.
IO	Interaction overview diagram.
SD	Sequence diagram.
T	Set of all integration tests and their respective path in the IO and SD diagrams.
T.IO	Ordered path list in the interaction overview diagram for each integration test case.
T.SD	Ordered path list per sequence diagram frame for each integration test case.
UT	Set of the unit tests and their targeted methods.
M	Set of changed methods.
SD.M	A list of methods involved per SD.
IO	Original interaction overview diagram.
IO'	Updated interaction overview diagram.
T'	Set of all tests from T selected for retest.
T"	Set of all tests from T marked as candidates for retest (before they are further refined.)
UT'	Set of all tests from UT selected for retest.
t	A single integration test case.
ut	A single unit test case.

4.2 Assumptions

1. SD messages are method calls. SD messages should be consistent with the called methods, so the message will carry the same name as the invoked method, the message will be referenced as: Object.MethodName().
2. Instrumentation is supposed to be used when running the tests in order to record the correct coverage path.
3. T is the set of integration test Cases, and the following tables are maintained using instrumentation.
    a. T.IO where for each $T_i \in T$, the table T.IO will contain the ordered traversal list of interaction overview artifacts.
    b. T.SD where for each $T_i \in T$, the table T.SD will contain the ordered traversal list of SD in the form of: OBJECT.METHOD.
4. UT is the set of unit test cases where each $UT_i \in UT$, UT_i.method specifies the method UT_i tests.
5. There is one interaction overview diagram for the entire integration.
6. A sequence diagram is identified by its unique name in the interaction overview diagram. Hereafter, referenced as the signature of the sequence diagram.
7. Upon updating methods in the source code, their version number in the class diagram should be updated as well.
8. A list of methods involved per SD, SD.M, is maintained before and after updating the IO diagram.

4.3 Description of Technique

Consider CD and CD', IO and IO', the set of integration (system) test cases: T, and the set of unit test cases UT. The technique for test case selection is mainly classified into two parts: the first is the detection and classification of changes based on the class diagram and interaction overview diagram, and the second is the classification of test cases based on their coverage in the interaction overview diagram. The two cases addressed in our technique for test case selection are:

I. Test case selection based on changes in class diagram: Firstly, our approach generates a set of changed methods, M from class diagram changes, as discussed in Chapter 3. Then we use M to generate the set of unit and integration tests that need to be re-tested.

   a. Generate a set of changed methods M: Changes in a class diagram affecting one or more UML entities (Association, Class, Method ...) Similar to the approach adopted in (Briand et al., 2002). We consider each class in both class diagrams, and compare the methods signature, contract, and version. Only if there is a change, we add this method to M.

   b. Select Integration Test cases: for every method in M, select from T.SD the test cases that have a reference call to the changed methods.

   c. Select Unit test cases:

      i. Select all the unit test cases that test the methods in M.

      ii. Select all the unit tests for methods that M is reachable from. In an SD, the flow of methods specifies which methods are called

before others, however, a change in method m, would imply that the methods calling m need to be tested as well. Therefore, for every SD, we consider all methods whose calls precede those in M in order to retest their unit tests. Our approach would be by generating the transitive closure (Goralcikiova and Konbek, 1979) for each method in every SD. The transitive closure output is a graph showing all the reachable methods from each other within the same sequence diagram. We then select all the methods for each SD where elements of M are reachable from.

II. Test case selection based on changes in the interaction overview diagram:

a. Compare every SD with the same signature in IO and IO' and mark all the changed SDs as changed. This is performed by doing a lifeline by lifeline and messages comparison between the SD in IO and that in IO'. If there is a change in one lifeline, we mark SD as changed.

b. Generate Tests Based on IO changes: Compare IO and IO', if there is a changed flow/edge, decision, decision final, or SD signature, then all test cases traversing the changed section are marked for retest. If an SD signature is not changed, but the SD itself was found marked changed, then we set the status of all the unclassified test cases that traverse those changed SDs marking them as candidates for retest. The comparison algorithm works by traversing IO and IO' in parallel.

c. Reduction: For each test case t marked as candidate for selection, do the following:

i. For every SD in the path of t marked as *CHANGED*, do the following:

   1. If there is no changed message, or order on the traversed path of t inside the SD, as per T.SD, then move to the next SD in the path list of t in T.IO.
   2. Otherwise, mark t for re-test.

4.4 Test Case Selection Algorithm

Based on the description of this technique, we shall divide the algorithm into two sections. In the first section we generate both unit and integration test cases for retest based on one for class diagram changes. In the second section we select only integration test cases based on interaction overview diagram changes.

4.4.1 Test Selection Based on Class Diagrams

We first generate the set of changed methods from the class diagram, and then perform selection of test cases from unit and integration level based on directly changed and indirectly affected methods.

```
Algorithm: CDTestCaseSelection.
Input:     T=Set of integration level tests.
           UT=Set of unit tests.
           M=Set of changed methods.
           IO=Original interaction overview diagram.
           IO'=Updated interaction overview diagram.
Output:    T'=Set of tests selected from T for retest.
           UT'=Set of tests selected from UT for retest.
Description: The algorithm generates two sets of test cases from UT and T for
             retest.
CDTestCaseSelection(T,UT, M, IO, IO'):T',UT'
  begin
  T'=∅  --set of all tests from T marked for retest
  T"=∅ --set of all tests from T marked as candidates
  M=GenerateChangedMethods(CD, CD')
  T'=CDIntegrationTestSelection(M, SD.M, T)
  UT'=CDUnitTestSelection(M, UT, IO')

  return T,UT
  end
```

Figure 6 Class Diagram Test Case Selection Algorithm

4.4.1.1 Directly Changed Methods in a Class Diagram

Figure 7 is the algorithm for generating the set of the directly changed methods, M, in the system. This algorithm is based on the class diagrams by comparing the signature of methods for each class and updating the list of changed methods M respectively.

```
Algorithm: GenerateChangedMethods.
Input:       CD=Original class diagram.
             CD'=Updated class diagram.
Output:      M=Set of changed methods.
Description: For each class in CD and its respective class in CD', compare the
             methods' signatures for changes.
CDIntegrationTestSelection (M, T):T'
  begin
  ∀ class c ∈ CD ∩ CD'
    begin
      ∀ method m ∈ c
        if the signature of m in CD is different from that in CD'
        --changed method
          M=M∪ {m}
    end
  return M
  end
```

Figure 7 Generate Directly Changed Methods Algorithm

The GenerateChangedMethods algorithm compares methods in the class diagram. GenerateChangedMethods has two loops, one for all classes and another for all methods inside each of these classes. The cost of this method is: $O(m)$ where m is the total count of all methods in the class diagram.

4.4.1.2 Integration Test Case Selection from Method Changes

Figure 8 is the algorithm for selecting integration test cases, T', for retest. It is based on the set of the directly changed methods, M, in the system. Every integration test case traversing methods in M should be selected for retest.

```
Algorithm: CDIntegrationTestSelection.
Input:      M=Set of changed methods.
            SD.M=List of methods in an SD.
            T=Set of integration test cases.
Output:     T'=Set of integration test cases selected for retest.
Description: Select all the SDs traversing each method in M (from SD.M). Then
             from T.SD, select all the integration test cases that traverse the selected
             SDs.
CDIntegrationTestSelection(M, SD.M, T)
  begin

  --For every integration test case
  --select only those that traverse the changed methods in M for retest.
  T'=∅ --set of all tests from T selected for retest
  ∀ t.sd ∈ T.SD
    --t.sd.path is the path traversed by a test case in an sd, from T.SD table
    if t.sd.path ∩ M ≠ Φ

  return T'
  end
```

Figure 8 Class Diagram Integration Test Selection

CDIntegrationTestSelection has one main loop that selects test cases from T.SD that traverse the changed methods. For ever row in T.SD we are checking if any element of M exist in the T.SD.path. This will cost $O(t.sd * l * m)$ where *t.sd* is the number of rows in T.SD, l is the average length of paths traversed in T.SD, and m is the number of all changed methods.

4.4.1.3 Unit Test Case Selection from Method Changes

Figure 9 is the algorithm for selecting unit test cases, UT', for retest. We first select the unit tests for the directly changed methods. Also, the algorithm selects the indirectly changed methods. This is done by generating the transitive closure for each method in every SD. The transitive closure graph shows all the reachable methods from each other. We use this transitive closure graph in identifying all the methods where elements of M are reachable from.

```
Algorithm: CDUnitTestSelection(M, UT, IO')
Input:      M=Set of changed methods.
            UT=Set of unit tests.
            IO'=Updated interaction overview diagram.
Output:     UT'=Set of unit tests selected from UT for retest.
Description: Select the unit test cases for methods in M and those whose calls
              precedes M in the same SD.
CDUnitTestSelection(M, UT, IO')
  begin

  UT'=∅ --set of all tests from UT marked for retest

  -- select the unit test cases for indirectly affected functions
  ∀ sd ∈ IO'
    begin

    --let G be the generated transitive closure for the SD considering the SD
    --as a directed graph with the messages as edges, and the methods and all the
    --other artifacts as nodes.
    G=generate_transitive_closure(SD)
    M'=∅ --set of all methods indirectly affected by methods in M
    ∀ node n ∈ G where n is a method and n∉ M'∪M
      ∀ m ∈ M
        if edge (n,m) ∈ G then
        --there exists a link between n and m, so select n for unit test traversal
          M'= M' ∪ {n}
    end

  --select the unit test cases for the directly and indirectly affected methods
  ∀ m ∈ M' ∪ M
      -- select the unit test cases from UT that test m
        UT'=UT'∪ {ut ∈ UT: ut is a test case that tests method m}

  return UT'
  end
```

Figure 9 Class Diagram Integration Test Selection

CDUnitTestSelection has two main loops, one to generate the indirectly changed methods for each SD and the other is for selecting unit test cases of the changed methods. The cost of generating the indirectly changed methods for each SD is $O(sd * (e * n + m))$ where e is the average edge count in the SDs, n is the average number of nodes in the SDs, and m is the number of changed methods. The $(e * n)$ is the worst case cost for generating the transitive closure graph for each SD. Selecting

the unit test cases from UT that test the changed methods costs $O(m' * ut)$ m' is count of all changed methods in the system (directly and indirectly). Therefore, CDUnitTestSelection costs $O(sd * (e * n + m)) + O(m' * ut))$.

4.4.2 Test Selection Based on Interaction Overview Diagrams

This algorithm is described in Figure 10. Since the IO diagram (Figure 3) is a planar graph, IO comparison is done by doing a breadth first traversal of both graphs in parallel starting from the start node. If there is a change detected in an IO artifact, the test cases traversing that changed artifact will be selected for retesting. However, if a test case passes through a changed SD, then that test case is marked for refinement.

Sequence diagrams (Figure 5) are planar graphs. Generating a set of changed SDs is done by comparing SDs with the same signature in IO and IO'. SDs (SD for IO and SD' for IO') are compared lifeline by lifeline. Lifelines are compared by ordered messages comparison between the lifeline in SD and its corresponding lifeline in SD'. If there is a change in one lifeline, we mark SD as changed.

```
Algorithm: IOTestCaseSelection.
Input:      T=Set of integration level tests.
            IO=Original interaction overview diagram.
            IO'=Updated interaction overview diagram.
Output:     T'=Set of tests selected from T for retest.
Description:  The algorithm generates a set of integration test cases to be selected for
              retest based on their traversal of a changed section in the IO diagram
SystemTestSelection(IO,IO',T):T'
  begin
    T'=∅ --set of all tests from T marked for retest based on IO changes
    T"=∅ --set of all tests from T marked as candidates
    ChangedSD=∅ --set of all changed sequence diagrams

    --compare the SDs of each interaction overview diagram
    ∀ sd ∈ IO having the same signature as sd' ∈ IO'
      begin
      ∀ ll ∈ sd ∪ sd'
        if ll does not exits in either sd or sd' then
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
        else
          begin
          if sd.ll and sd'.ll have different messages or message order
          --mark sd as changed
          ChangedSD=ChangedSD ∪ {sd}
          end
      end

    --traverse the IO diagram and generate a set of test cases for retest
    --and another as candidates for refinement in SDBasedReduction
    (T',T")=IOBasedClassification(IO, IO', T, ChangedSD)
    T'=T' ∪ SDBasedReduction(T, T", ChangedSD)
  end
```

Figure 10 System Test Case Selection Algorithm

IOTestCaseSelection will first compare every SD diagrams in IO with its respective diagram in IO'. Then, we will call two methods: IOBasedClassification followed by SDBasedReduction. Comparing all the SDs of each interaction overview diagram with each other costs $O(sd * a)$ where *sd* is the count of all the SDs in IO and *a* is the average number of SD artifacts in the SDs. The costs of calling IOBasedClassification and SDBasedClassification will be discussed in sections 4.4.2.1 and 4.4.2.2 respectively.

4.4.2.1 Interaction Overview Diagram Based Classification

IOBasedClassification algorithm, presented in Figure 11, will perform test case selection based on the interaction overview diagram before and after the update. This algorithm will traverse the original and updated IO diagrams in parallel detecting changes along the path and selecting test cases that traverse that change. Based on the change type, the algorithm will either classify the test case for retest or as candidates for further analysis and reduction. Candidates are usually test cases that traverse a changed SD. This is because if an SD is internally changed, test cases traversing that SD may not traverse the change inside that SD.

```
Algorithm: IOBasedClassification.
Input:      T=Set of integration level tests.
            G=Node or branch in the original interaction overview diagram.
            G'=Node or branch in the updated interaction overview diagram.
            ChangedSD=set of changed SDs
Output:     T'=Set of tests selected from T for retest.
            T"= Set of tests selected from T as candidates for retest.
Description: The interaction overview based classification algorithm it starts at the start node of both
interaction overview diagrams; and start traversing the edges and nodes. If an edge is deleted or
changed, all the test cases having that edge in their path will be marked for retest. Similarly if there is a
change in the decisions. If a test case passes through a changed SD, then that test case is marked for
refinement.
IOBasedClassification (G, G', T): T', T"
begin
  if G is marked as visited
    return; -- no need to go into loops
  else
    mark G as visited
  if signature(G)=signature(G') then -- same node
    begin
    switch G.type:
    case action:
            if (G ∈ ChangedSD)
                --G is a changed SD with the same signature
                --mark all test cases traversing G as candidates for selection
                ∀ t ∈ T with G in the path of t.IO, T'=T"∪ { t }
            [T',T"]= IOBasedClassification (G.outedge, G'.outedge, T)
   case edge:
            --move to the targets and continue processing
            [T',T"]= IOBasedClassification (G.target, G'.target, T)
   case decision:
            --mark all test cases traversing G as selected for retest
            ∀ decision d of G
                if d ∈ decision(G')
                    [T',T"]= IOBasedClassification (G. d, G'. d, T)
                else
                -- deleted decision, mark all test cases traversing this decision as
                -- selected for retest
                    begin
                    ∀ t ∈ T with d in the path of t.IO do
                        T'=T'∪ { t } --Retest
                        T"=T"-{ t } --in case already marked as Candidate
                    end
   default: return [T',T"]
   end
  else
   begin
   ∀ t ∈ T with G in the path of t.IO do
            T'=T'∪ { t } -- retest
            T"=T"-{ t } -- in case already marked as Candidate
   end
end
```

Figure 11 IO Based Classification/Selection Algorithm

IOBasedClassification, is a graph coverage algorithm, it costs $O(e * t * l)$ where e is the number of edges in *IO*, t is the number of integration test cases in T.IO, and l is the average length of path per test case in T.IO. That is due to the fact that IOBasedClassification traverses every artifact in IO, and for every changed artifact searches for test cases that traverse that artifact. The $(t * l)$ is actually the input path in T.IO table; hence, we only traverse the T.IO table once per changed IO artifact.

4.4.2.2 Sequence Diagrams Based Reduction

After the test cases have been classified based on the interaction overview as candidates for further refinement, the SD-based reduction algorithm (Figure 12) refines that classification selecting a reduced set of integration level test cases for retest. For each candidate test case, consider the changed SDs that case traverses, we will check that the SD path covered by each candidate test case in T.IO is still valid in the changed SD.

Algorithm: SDBasedReduction.
Input: T=Set of integration level tests.
T"= Set of tests selected from T as candidates for retest.
ChangedSD=set of changed SDs
Output: T'=Set of tests selected from T for retest.
Description: The SD based classification algorithm goes through each SD classified from T, T" and check the traversal flow if changed between IO and IO' at the SD-level.

```
SDBasedSelection (T, T"): T'
begin
  ∀ t ∈ T"
    begin
    ∀ sd ∈ (SDBasedSelection ∩ set of SDs in the path of t.IO)
      --sd is marked as changed, so go to sd level
      begin
      if t.sd path can be traversed in SD-level path in IO'.sd
        continue
      else
        --mark t for retest
        begin
        T'=T'∪ { t } (Retest)
        T"=T"-{ t } (in case already marked as Candidate)
        end
      end
    end
end
```

Figure 12 SD Based Classification/Selection Algorithm

SDBasedClassification will compare method path per changed SD per candidate test case. The cost for that is $O(t'' * sd * l)$ where t'' is the number of candidate tests of retest, sd is the total number of changed SDs, and l is the average method-call path traversed per SD from T.SD. The $(sd * l)$ is the input path in T.SD table; hence, we only traverse the T.SD table once per candidate.

Chapter 5

Case Study

We will test our test case selection technique against an ATM application as part of our case study. Figure 1 is the class diagram of this example. This application handles ATM (Automated Teller Machine) and POS (Point of Sale) transactions. In section 5.1, we will identify the changes on the class and interaction overview diagrams. We will provide the pool of unit (UT) and integration (T) test cases in section 5.2, that we will apply our technique on and present the results in section 5.3.

5.1 Changes Tracking

We track the changes in the class diagram, sequence diagrams, and interaction overview diagrams. The ATM example presented in Figure 1, has changes in the class diagram (section 5.1.1), sequence diagram (section 5.1.2), and interaction overview diagram (section 5.1.3).

5.1.1 Changes in the Class Diagram

As per the design changes, a new functionality have been added to the system to validate monthly limit.

Affected classes (by design comparison):

Card class:

Attributes:

New Attributes:

monthlyLimitUsed

monthlyLimitLastDate

Methods:

New Methods:

updateMonthly(int, float)

CardType class:

Attributes:

New Attributes:

monthlyLimit

Methods:

New Methods:

getMonthlyLimit

Affected classes (by change of contract):

BackOffice class:

Methods:

Changed Methods:

validateLimit

Affected classes (by change of contract):

BackOffice class:

Methods:

Changed Methods:

validateLimit

5.1.2 Changes in the Sequence Diagrams

As we consider all the SD's in the system, namely the cash withdrawal (Figure 5), Balance Inquiry, Print Receipt, Dispense Cash, purchase Through POS, Reservation of Money (Figure 13), POS Print Receipt.

In SD Reservation of Money (Figure 13) has been changed to the SD in Figure 14 by re-ordering the recordTransaction(), thus the lifeline BackOffice was changed by adding a method call to recordTransaction(). Account lifeline was also changed by removing the method call for recordTransaction(), the lifeline TransactionLog was also removed.

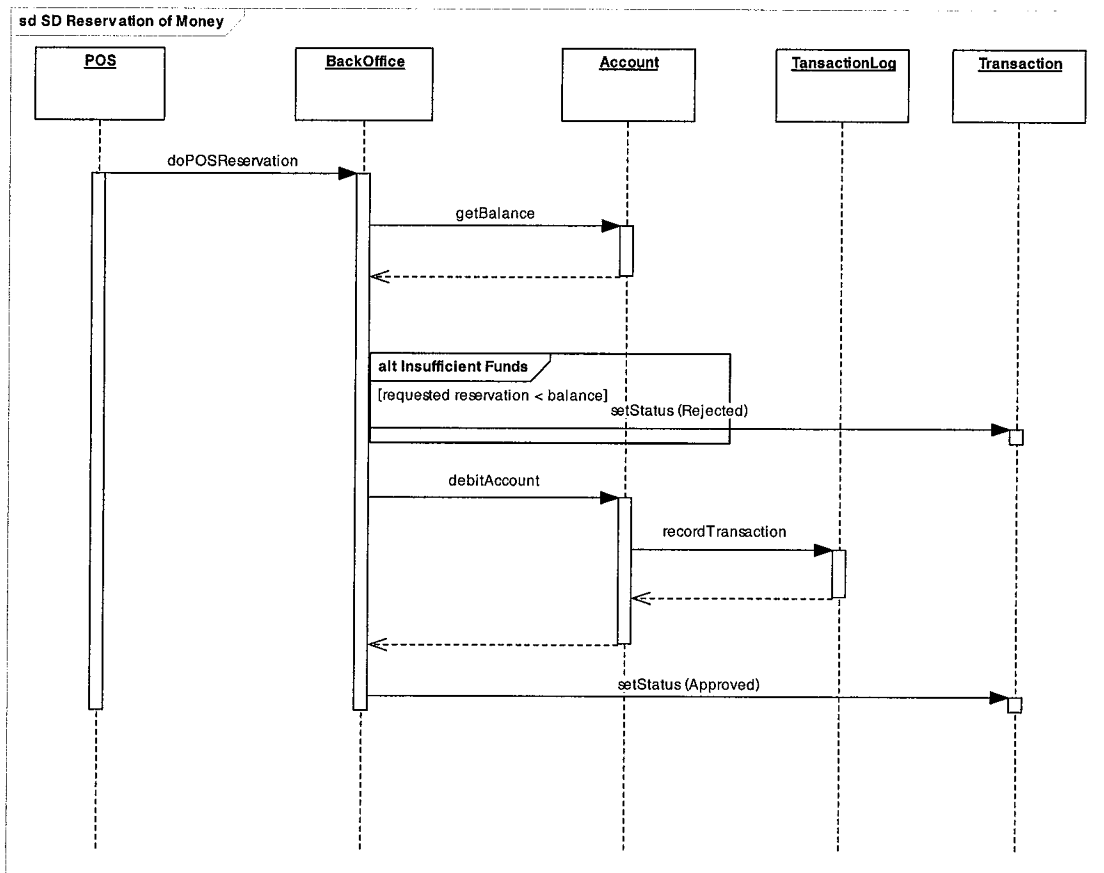

Figure 13 SD Reservation of Money

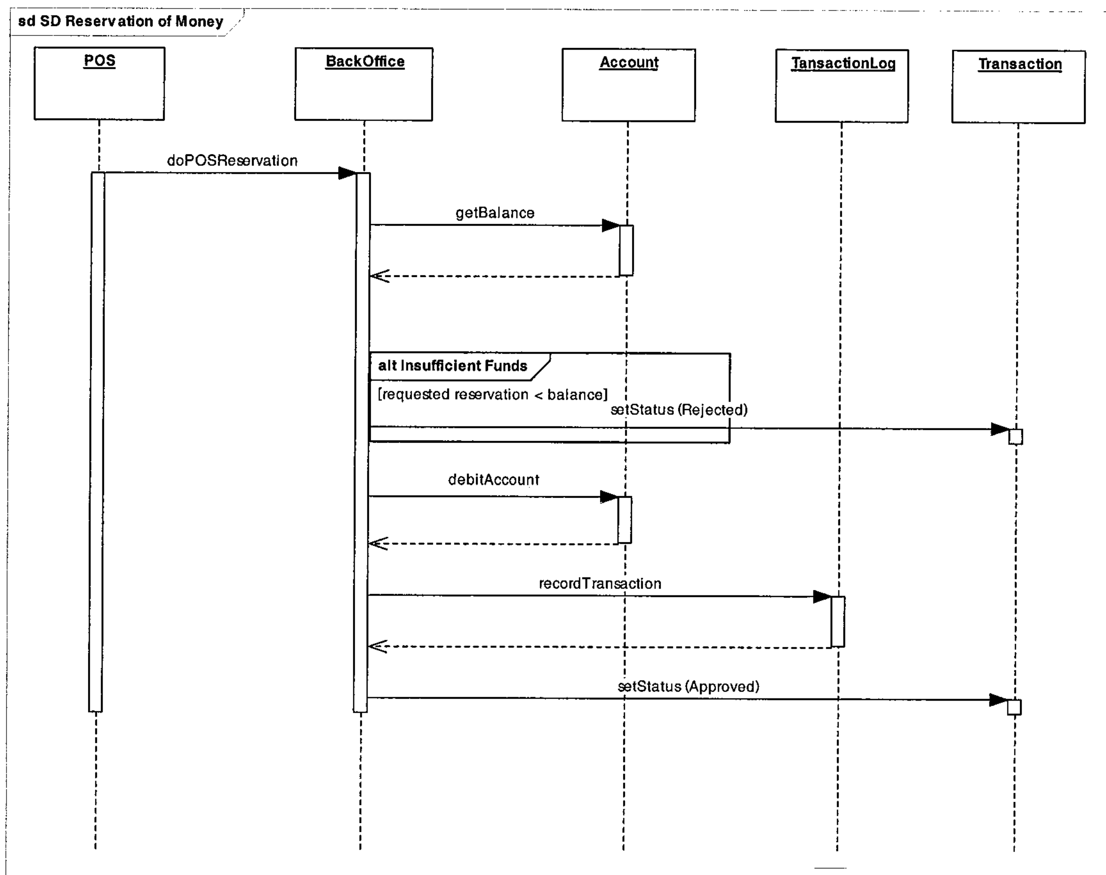

Figure 14 SD Reservation on Money after update

5.1.3 Changes in Interaction Overview Diagrams

A5 was introduced along with the corresponding control flows: F11, F7, and F12. This resulted in deletion of control flows F6 and F8. See Figures 3&4.

5.2 Original Suit of Test Cases:

The integration level test cases, with their respective path in the interaction overview diagram, (T.IO) are presented in Table 2. The path of each integration test case inside the sequence diagrams (T.SD) is presented in Table 3. Finally, Table 4 contains a list of unit test cases for the methods of the ATM example.

5.2.1 Integration Test Cases Path in the Interaction Overview Diagram.

Table 2 lists the integration test cases path in the interaction overview diagram of the atm example (T.IO).

Table 2 ATM example integration test cases path in the interaction overview diagram (T.IO)

Test Case	Description	Path
T1	ATM Valid balance inquiry	F13, D4, F1, A1, F2, D1, F8, A3, F9, D3, F11, A5, F12
T2	ATM Valid cash withdrawal	F13, D4, F1, A1, F2, D1, F3, A2, F4, D2, F6, A4, F7, A5, F12
T3	ATM Invalid balance inquiry, invalid pin.	F13, D4, F1, A1, F2, D1, F8, A3, F9, D3, F10
T4	ATM Invalid cash withdrawal, invalid pin.	F13, D4, F1, A1, F2, D1, F3, A2, F4, D2, F5
T5	ATM Invalid cash withdrawal exceeds daily limit.	F13, D4, F1, A1, F2, D1, F3, A2, F4, D2, F5
T6	ATM Invalid cash withdrawal, insufficient funds	F13, D4, F1, A1, F2, D1, F3, A2, F4, D2, F5
T7	POS Valid reservation	F13, D4, F14, A6, F15, D5,F16, A8, F19, D7, F23
T8	POS Valid purchase	F13, D4, F14, A6, F15, D5, F17, A7, F18, D6, F22, A9, F24
T9	POS Invalid reservation, insufficient balance	F13, D4, F14, A6, F15, D5,F16, A8, F19, D7, F21
T10	POS Invalid purchase insufficient balance	F13, D4, F14, A6, F15, D5,F17, A7, F18, D6, F20
T11	POS Valid reservation, exact balance	F13, D4, F14, A6, F15, D5,F16, A8, F19, D7, F23
T12	POS Valid purchase, exact balance	F13, D4, F14, A6, F15, D5, F17, A7, F18, D6, F22, A9, F24

5.2.2 Integration Test Cases Path in the Sequence Diagrams.

Table 3 lists the integration test cases path in the sequence diagrams of the atm example (T.SD).

Table 3 ATM example integration test cases path in the sequence diagrams (T.SD

Test Case	SD	Path
T1	A3	ATM.doBalanceInq, Backoffice.ValidatePin, Backoffice.ValidatePin.Return, [alt: invalid pin], BackOffice.logTransaction, BackOffice.logTransaction.Return, BackOffice.setStatus
T2	A2	ATM.doCashWithdrawal, Backoffice.ValidatePin, Backoffice.ValidatePin.Return,[alt: invalid pin], Backoffice.validateLimit, Backoffice. validateLimit.Return, [alt: exceed limit], Backoffice.ValidateBalance, Backoffice.ValidateBalance.Return, [alt: insufficient funds], BackOffice.logTransaction, BackOffice.logTransaction.Return, BackOffice.setStatus
T2	A4	ATM.dispenseCash, ATM.Return
T3	A3	ATM.doBalanceInq, Backoffice.ValidatePin, Backoffice.ValidatePin.Return, [alt: invalid pin], BackOffice.setStatus
T4	A2	ATM.doCashWithdrawal, Backoffice.ValidatePin, Backoffice.Return, [alt: invalid pin], BackOffice.setStatus
T5	A2	ATM.doCashWithdrawal, Backoffice.ValidatePin, Backoffice.ValidatePin.Return, [alt: invalid pin], Backoffice.validateLimit, Backoffice. validateLimit.Return, [alt: exceed limit], BackOffice.setStatus
T6	A2	ATM.doCashWithdrawal, Backoffice.ValidatePin, Backoffice.ValidatePin.Return, [alt: invalid pin], Backoffice.validateLimit, Backoffice. validateLimit.Return, [alt: exceed limit], Backoffice.ValidateBalance, Backoffice.ValidateBalance.Return, [alt: insufficient funds], BackOffice.setStatus
T7	A8	BackOffice.doPOSReservation, Account.getBalance, Account. getBalance.Return,[alt:insufficient funds], Account.debitAccount, TransactionLog.recordTransaction, TransactionLog. recotdTransaction.Return, Account.debitAccount.Return, Transaction.setStatus
T8	A7	BackOffice.doPOSPurchase, Account.getBalance, Account. getBalance.Return,[alt:insufficient balance], Account.withdrawFromAccount, Account.withdrawFromAccount.Return, TransactionLog.logTransaction, TransactionLog.logTransaction.Return, Transaction.setStatus
T8	A9	POS.rPrintReceipt

T9	A8	BackOffice.doPOSReservation, Account.getBalance, Account.getBalance.Return,[alt:insufficient funds], Transaction.setStatus
T10	A7	BackOffice.doPOSPurchase, Account.getBalance, Account. getBalance.Return,[alt:insufficient balance], Transaction.setStatus
T11	A8	BackOffice.doPOSReservation, Account.getBalance, Account. getBalance.Return,[alt:insufficient funds], Account.debitAccount, TransactionLog.recordTransaction, TransactionLog. recotdTransaction.Return, Account.debitAccount.Return, Transaction.setStatus
T12	A7	BackOffice.doPOSPurchase, Account.getBalance, Account. getBalance.Return,[alt:insufficient balance], Account.withdrawFromAccount, Account.withdrawFromAccount.Return, TransactionLog.logTransaction, TransactionLog.logTransaction.Return, Transaction.setStatus
T12	A9	POS.rPrintReceipt

5.2.2 Integration Unit Test Cases.

Table 4 lists the unit test cases for the methods of the ATM example.

Table 4 Unit Test Cases (UT)

UT1	Card.getAccount()
UT2	Card.getCardType()
UT3	Card.updateDaily()
UT4	Card.getExpDate()
UT5	Card.updateAccount()
UT6	ATM.createTransaction()
UT7	POS.getLocation()
UT8	POS.setLocation()
UT9	POS.printAndExit()
UT10	POS.postPurchase()
UT11	POS.getCardBalance()
UT12	POS.doPOSPurchase()
UT13	POS.doPOSReservation()
UT14	ATM.dispenseCash()
UT15	ATM.printReceipt()
UT16	POS.printReceipt()
UT17	Account.SetBalance()
UT18	Account.SetStatus()
UT19	Account.SetCurrency()
UT20	CardType.getCardLife()
UT21	CardType.getDailyLimit()
UT22	TransactionLog.logTransaction()
UT23	TransactionLog.updateLog()
UT24	CardType.getMontlyLimit()
UT25	Card.updateMontly()
UT26	BackOffice.validateLimit()
UT27	POS.rPrintReceipt()
UT28	ATM.returnCard()
UT29	ATM.getTranactionRequest()
UT30	ATM.dispenseCash()
UT31	ATM.printAndExit()
UT32	Transaction.getCard()
UT33	Transaction.getStatus()
UT34	Transaction.setStatus()
UT35	HSM.checkPin()
UT36	BackOffice.validatePin()
UT37	BackOffice.validateBalance()
UT38	BackOffice.doBalanceInq()
UT39	BackOffice.doCashWithdrawal()
UT40	BackOffice.doErrorExit()
UT41	Account.GetBalance()
UT42	Account.GetStatus()
UT43	Account.GetCurreny()

5.3 Test Case Selection

We apply the algorithm presented in Chapter 4 on the ATM example using the changes introduced in section 5.1.1, 5.1.2, and 5.1.3. The original set of test cases is presented in section 5.2.

5.3.1 Test Case Selection Based on Class Diagram

Assume the change of the class diagram is in the method validateLimit().

There fore, the set of changed methods, M= {validateLimit()}

The SD referencing M is A2 (SD CashWithdrawal, Figure 5).

The integration test cases traversing A2 on the IO level from T.SD (Table 3) are: T2, T5, & T6.

The unit test cases testing M directly (Table 4) is UT26.

The indirectly affected methods to M are: validatePin() and doCashWithdrawal().

The unit test cases for the methods preceding those of M are: UT36 and UT39.

Therefore, the selected test cases for regression testing based on changes in SD are: T2, T5, T6, UT26, UT36, and UT39

5.3.2 Test Case Selection Based on Changes in the Interaction Overview Diagram

The set of changed SDs based on comparison between IO and IO' is: A8 (SD Reservation of Money, Figure 13 and Figure 14).

Assume that the Direct Change in the IO diagram: deletion of edges F6 and F8 and the introduction of control flows F11, F7, and F12. (Figure 4)

The set of test cases from T.IO selected for retesting are those traversing F6 is T2.

The set of candidate test cases traversing A8 (SD Reservation of Money) is: {T7, T9, T11}

After checking if the path list of the these candidates in their respectively changed SDs in T.SD (Table 3) the results are:

(T7,A8)= path invalid. *(BackOffice.doPOSReservation, Account.getBalance, Account. getBalance.Return,[alt:insufficient funds], Account.debitAccount, TransactionLog.recordTransaction, TransactionLog. recotdTransaction.Return, Account.debitAccount.Return, Transaction.setStatus)* because there is no immediate path between TransactionLog.recordTransaction and TransactionLog. recotdTransaction.Return.

(T9,A8)= path is valid. *(BackOffice.doPOSReservation, Account.getBalance, Account.getBalance.Return,[alt:insufficient funds], Transaction.setStatus).* That means that T9 should not be selected for retest based on the SD Return Cash because it doesn't traverse the changed section.

(T11,A8)= path invalid. *(BackOffice.doPOSReservation, Account.getBalance, Account. getBalance.Return,[alt:insufficient funds], Account.debitAccount, TransactionLog.recordTransaction, TransactionLog. recotdTransaction.Return, Account.debitAccount.Return, Transaction.setStatus)* because there is no immediate path between TransactionLog.recordTransaction and TransactionLog. recotdTransaction.Return.

Therefore, the selected test cases for regression testing based on changes in IO are: T2, T7, and T11.

Chapter 6

Conclusion and Future Work

The work presented in this paper selects unit and system tests for regression testing. This test case selection technique is based on the design without access to source code and the logic is derived from the sequence diagrams. We used class diagrams and interaction overview diagrams as the basis of our unit and integration test selection technique.

We have empirically presented a case study on ATM and POS transaction example. Our case study showed that this technique selects a limited number of tests that covers the specific changes. We changed one method, one IO entity, and an order of execution inside an SD, our technique selected 3 unit tests and 5 interaction tests from as test suit consisting of 43 unit level tests and 12 integration level tests.

Finally, in future works, we can perform further empirical studies on this approach. We can also enhance the algorithm making use of dependency information gathered from the Object Constraint Language (OCL) in the class diagram to gain precision. An improvement would be to handle more than one class and interaction overview diagrams.

Bibliography

Agrawal, H. Horgan, J.R. and Krauser, E.W., 1993. Incremental regression testing. In: *Preceedings of the Conference on Software Maintenance*, 348-357.

Arlow, J. and Neustadt, I. (2005) *UML2 and the Unified Process Second Edition*, USA: Addison-Wesley.

Born, M., Schieferdecker, I., Li, M. UML Framework for Automated Generation of Component-Based Test Systems. In: SNPD.

Briand, L. C., Labiche, Y., Buist, K. and Soccar, G. (2002) Automating Impact Analysis and Regression Test Selection Based on UML Designs. In: Proceedings of the IEEE International Conference on Software Maintenance, Montreal.

Cavarra, A., Crichton, C. and Davies. J. (2004) A method for the automatic generation of test suites from object models. In: *Information & Software Technology*, 46(5), 309-314.

David, C. K., Gao, J. and Chen, C. (1996) On regression testing of object-oriented programs. In: *The Journal of Systems and Software,* 32, 21-40.

Fowler, M. (2003) *UML Distilled Third Edition*, USA: Addison-Wesley.

Fraikin, F. and Leonhardt, T. (2002) SeDiTeC –Testing Based on Sequence Diagrams. International Conference on Automated Software Engineering, 261-266.

Goralcikiova, A. and Konbek, V. (1979)A reduct and closure algorithm for graphs. In: *Mathematical Foundations of Computer Science*, 301-307, Springer Verlag, Lecture Notes in Computer Science V. 74.

Gupta, R., Harrold, M.J., Soffa, M.L., 1996. Program slicing-based regression testing techniques. Software Testing, Verification and Reliability 6 (2), 83-111.

Harrold, M.J., Gupta, R., Soffa, M.L., 1993. A methodology for controlling the size of a test suite. ACM Trans. Software Eng. And Methodology, July, 270-285.

Hsia, P., Li, X., Kung, D., Hsu, C., Li, L., Toyoshima, Y., and Chen, C. A Technique for the Selective Revalidation of OO Software (1997) Software Maintenance: Research and Practice, (9) 217-233

Kansomkeat, S., and Riverpiboon, W. (2003) Automated-Generating Test Case Using UML Statechart Diagrams. Proceedings of SAICSIT,296-300.

Mansour, N., Bahsoon, R., and Baradhi, G. (2000) Empirical comparison of regression test selection algorithms. The Journal of Systems and Software, (57), 79-90.

Mansour, N. and Bahsoon, R. (2002) Reduction-based methods and metrics for selective regression testing. Information and Software Technology, 44 (7). 431-443.

Mansour, N., El-Fakih, K., 1999. Simulated annealing and genetic algorithms for optimal regression testing. J. Software Maintenance 11, 19-34.

Offutt, J. and Abdurazik, A. Using UML collaboration diagrams for static checking and test generation. In: 3rd International Conference on the UML, 383–395.

Offutt, J. and Aburazik, A. (2003) Generating Tests from UML Specifications. Software Testing, Verification and Reliability, 13(1), 25-53.

Rothermel, G., Harrold, M. J. and Dedhia, J. (2000) Regression test selection for C++ software. In: Journal of Software Testing Verification and Reliability, 10, 77-109.

White, L., Narayanswamy, V., Friedman, T., Kirschenbaum, M., Piwowarski, P., Oha, M., 1993. Test manager: a regression testing tool. In: Proceedings of the Conference on Software Maintenance, 338-347.

Wu, Y., Chen. and M., Offut, (2003)UML-Based Integration Testing for Component-Based Software. In: ICCBSS, 251-260.

Wu, Y., Chen, M. and Kao, H (1999) Regression Testing on object-Oriented Programs Tenth International Symposium on Software Reliability.