An Automatic Test Suite Regeneration Technique Ensuring State Model Coverage Using UML Diagrams and Source Syntax

Afrina Khatun* and Kazi Sakib†
Institute of Information Technology, University of Dhaka, Bangladesh
*bit0411@iit.du.ac.bd, †sakib@iit.du.ac.bd

***Abstract*—Automated test regeneration intends to ensure high coverage of the system model from an existing test suite. While regenerating test suite, most of the existing techniques ignore coverage achieved so far by existing test suite. An automatic test regeneration technique to achieve high coverage of state model is proposed in this paper which uses coverage result as well as information from UML diagrams and source code. The architecture of the technique is supported by three modules. The first module processes inputted UML diagrams, source code and test suite as XML elements, source class and test steps respectively. The second module measures model coverage result achieved by existing test suite. The final module combines coverage result, extracted UML and source syntax together to regenerate test cases. A case study has been conducted to assess the technique effectiveness and has been successful to regenerate unit and integration test cases that achieve full coverage of the state model.**

Keywords*—software testing, automatic test regeneration, model coverage analysis, unit and integration testing

I. Introduction

Test suite regeneration from existing test suite is done to ensure high test coverage of a System Under Test (SUT). Test coverage is a quality assurance metric which indicates how thoroughly a test suite exercises a given system [1]. A system model describes the behavior and requirements of the system. If a test suite fails to achieve full coverage of the system model, it indicates the test cases have missed important requirements of the SUT. In model based coverage analysis techniques, manual or auto generated test suite is executed against the system UML diagrams like - class, state, sequence diagrams and GUI screens. In this approach, coverage result is measured by parsing the test cases and finding corresponding elements in the UML diagrams. The coverage result is represented to the tester in a graphical or documented form which can play a vital role in test suite regeneration.

Achieving full coverage of the model generally fails, as test automation tools lack predefined coverage checking while generating test suites. Existing coverage analysis tools measure the coverage of a model, based on some predefined coverage criteria like - state coverage, transition coverage, all path coverage etc. These tools only identify the tested and untested elements of the system model. Therefore, to ensure coverage either existing test generation algorithms need to be changed or regeneration needs to be done. On the other hand, most of the automated model based test generation techniques generate test cases from an abstract model of the SUT [2]. However, those fail to find out how much coverage is achieved through the generated test suites. As a result, many system requirements may remain untested. Even if the number of uncovered elements is small, this may lead to manual inspection of the whole model again to find those elements.

A dependence graph-based test coverage analysis technique for object oriented programs has been proposed by Najumudheen et al. [3]. In this paper source code was instrumented and a call based system dependence graph was constructed by parsing source code to measure traditional coverage criteria like - statement, branch, method coverage etc. Therefore, the approach does not generate test cases for the uncovered part of the system graph. A model based test coverage analysis technique of UML state machines has been proposed by Ferreira et al. [4]. The proposed tool received XML formatted state model and auto generated test suite to identify covered and uncovered elements by the test suite. Therefore, the test suite still lacks test cases that are required to ensure the full coverage of the model. A technique to create test cases from UML models is offered by Sarma et al. [5]. In this technique UML use case and sequence diagram have been considered for generating test sequences. A test case regeneration approach based on sequential pattern mining has been proposed by Wei He et al. in [6] to produce test cases from existing test repositories. It uses a GA based approach to regenerate test cases from existing test repository. However, both of these techniques ignore coverage criteria of system model elements.

This paper incorporates coverage analysis result with test regeneration technique to produce unit and intra class integration test cases which ensure high coverage of the system state diagram. The technique contains three modules to manage the whole regeneration process. The *Parser*, *Coverage Analysis* and *Test Regeneration* - each module has some predefined responsibilities to support the technique implementation. The *Parser* is the input data provider of the proposed technique. It extracts test steps, source syntax and UML elements by parsing an existing test suite, source code and UML diagrams respectively and processes these information in a structured way to be used later. The *Coverage Analysis* module uses the processed test suite, UML class and state elements to identify the covered and uncovered elements of the state model by simulating the execution of test cases against the model. The *Test Regeneration* first uses the coverage result to generate all possible uncovered transition paths from the state diagram of the system. It then regenerates unit and intra class integration test cases for the generated uncovered transition paths. While regenerating tests, event call signature of transitions and method call syntax of source code are also considered to check consistency, that ensures regeneration of executable test cases.

A case study on a sample java project illustrating an ATM system [7], has been conducted here for the initial assessment

978-1-5090-1269-5/16/$31.00 ©2016 IEEE

of the proposed technique. The corresponding UMLs of the sample project were also built. Test suites of the sample project was generated using an existing automated test generation framework [8]. These source code, test suites and UMLs were extracted into source class, test steps and XML elements respectively by the *Parser*. The *Coverage Analysis* module received the XML data and test cases and produced the coverage result of the state model. These coverage result and source classes were received by the *Test Regeneration* module and test scripts were produced which achieved full coverage of the model.

II. Related Work

In the literature survey, several automated coverage analysis and test regeneration techniques have been proposed. Most of the techniques emphasize only one of the tasks of either test regeneration [6] or coverage measurement [4]. Some of the significant works related to this research topic are described in the following part.

Najumudheen et al. proposed a dependence graph based test coverage analysis technique for object oriented projects [3]. In this approach the source program is converted into a dependence graph based representation, named call based object oriented system dependence graph and instrumented at specific points. During the test execution, a coverage analysis is done based on some coverage criteria like - statement, branch, path, method coverage and the edges of the graph are marked as covered by a graph marker. However if test regeneration could be done for the unmarked edges of the graph, high coverage could have been achieved.

A state based coverage analysis and behavioral equivalence checking for C++ programs has been proposed by Heckeler et al. [9]. The approach transforms the UML state model to a transition table to detect behavioral deviation from the C++ implementing finite state machine. After transformation, it links the source code with appropriate states and executes existing unit, system and integration tests to find out which states are covered. However, the approach uses manual instrumentation and considers only state coverage. Considering transition sequence coverage along with state coverage could ensure high integration coverage of the integration tests. Again combining test regeneration technique with the coverage analysis process could produce more accurate test suites.

Ferreira et al. has proposed a state model based test coverage analysis tool [4]. The tool is developed as part of a GUI testing environment. The tool receives a system model and a test suite in XMI format as input. It then simulates the execution of the test suite over the model to determine the coverage achieved through the test suite. It represents the result in a colored UML state machine model to the tester. However the tool ends its task by only concluding the incompleteness of the test suite. Regenerating test case for the uncovered elements of the state model would make it more useful for the tester.

An integration testing coverage tool for object oriented program has been proposed in [10]. The tool automatically instruments the source code and collects coverage data while test execution. After execution, it identifies the uncovered methods from coverage result and creates a scenario graph using sequence and use case diagrams. It then extracts inter class method call path from the scenario graph that contains the uncovered method and generates test case for the path. However, the approach does not consider unit and intra class integration tests and also generates inexecutable test cases.

Few test case regeneration techniques have been also proposed in the literature survey. A sequential pattern mining based test regeneration technique for object oriented projects is presented by Wei et al. [6]. The approach applies sequential pattern mining strategy to obtain frequent subsequences of method call from existing test suite. It uses a GA based approach to regenerate test cases by mining the frequent subsequences but ignore coverage consideration. Alshahwan et al. proposed two test regeneration techniques for web applications [11] using standard and value based Def-Use testing accordingly. This approach combines HTTP requests from a test suite to form client side requests and combine fragment of these requests to regenerate test cases to execute server side requests. However, combining coverage criteria along with the regeneration process could produce more effective test cases.

Fraser et al. has proposed a tool called, EvoSuite for automatic test generation of object oriented programs [12]. However, it does not consider model requirements and coverage for test generation. An automatic test generation technique to detect operational, use case dependency and scenario faults has been proposed by Srama et al.[5]. The technique generates test cases from use case and sequence diagrams. However, the test generation process totally ignores source syntax information. Nahar et al. has proposed a framework for automatic test generation using software semantics and source syntax [8]. However it totally ignores coverage criteria while generating test cases. Chen et al. has proposed a technique for automatic test case generation for UML activity diagrams [13]. This approach executes random generated test cases against the UML activity diagram and selects test cases that satisfy predefined coverage criteria. Instead of reducing the test suite, regenerating new test cases ensuring the predefined coverage could make the test suite more accurate.

Most of the existing coverage analysis techniques in the literature only focus in identifying tested and untested elements and ignore test regeneration (e.g.[4]). Some test regeneration techniques are also available in the literature (e.g. [6], [11]). However, they do not ensure full coverage of a system state model. So even if the number of untested elements is small, it leads to manual inspection of the whole model.

III. Methodology

In this section, a technique for automated test regeneration ensuring state model coverage is proposed. As mentioned in the previous section, the existing approaches either regenerate test cases or measure coverage, but do not regenerate test cases from existing test suite, considering the state model coverage for intra class integration test. To meet up the limitations of the above individual approaches, an automated test suite regeneration technique, considering the coverage analysis result, and using UML diagrams and source syntax is proposed. The overview and the internal architecture of the proposed technique are described in the following part.

A. Overview of the Proposed Test Regeneration Technique

The top level overview of the proposed technique artifacts is shown in Fig. 1. The artifacts perform core tasks for the functionality of the technique. The *Source Code Parser*, *Test Suite Parser* and *UML Parser* shown in Fig. 1 process the inputted data into a structured form. This modules extract information from the source code, test suite and UML class, state diagrams which are provided by the user. *Coverage Analyzer* of Fig. 1 considers each test step (test statement) of a test case which is a method call event along with parameter values, and identifies the covered and uncovered elements of the state model by the existing test suite. *Test Regenerator* illustrated in Fig. 1 performs the task of test regeneration. It uses the coverage result, extracted UML elements and derived method syntax by the *Source Parser* to regenerate unit and integration test cases, for uncovered states, transitions and intra class method call sequences that is transition sequences.

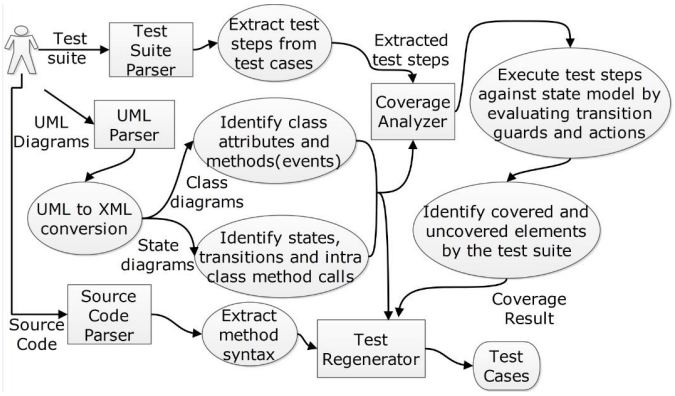

Fig. 1: Top Level View of the Test Regeneration Technique

B. Internal Architecture of the Test Regeneration Technique

The internal architecture of test regeneration technique shown in Fig. 2 consists of several modules, as each module performs different responsibilities. The architecture is divided into three major modules: (1) *Parser Module*, (2) *Coverage Analysis Module* and (3) *Test Regeneration Module*. Each module consists of several components which support its functionality. The functionality of these modules is described in the following part.

1) Parser Module: The *Parser Module* acts as input data provider for the proposed technique. It receives XML formatted UML diagrams, source code and test suite files from a user defined location. It contains three components as illustrated in Fig. 2. The *Source Syntax Identifier* component extracts class, method syntax from the existing source code files, and passes them to *Test Regeneration* module. On the other hand, *Test Steps Identifier* component shown in Fig. 2 extracts test steps from the test cases of existing test suite files. It identifies each test step that is method call statement along with its parameter values and passes those to the *Coverage Analysis* module.

UML Element Identifier receives XML formatted UML diagrams, and identifies class attributes, their initial values, methods, parameter list from class diagrams. As each state machine is usually associated to a class, the class attributes represent state variables in the state machine, and the class

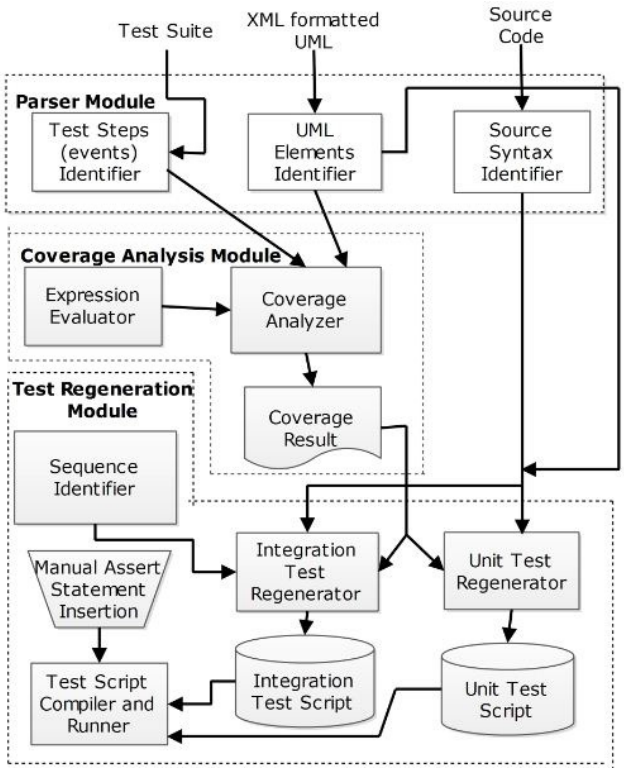

Fig. 2: Internal Modules of the Test Regeneration Technique

methods define the signature of call events in the state machine. The information of states, transitions, corresponding guards and action events are also extracted from state diagrams. This information is used later by the *Coverage Analysis* and *Test Regeneration* module as showed in Fig. 2 to measure model coverage and regenerate unit and integration tests.

2) Coverage Analysis: *Coverage Analysis* module is responsible for the computation of the coverage achieved by existing test suite. Coverage criteria must be measurable and be an indicator of the test adequacy. One of the coverage criteria applicable to system state models is transition based coverage. All states and transitions coverage represent the full coverage of all available methods of the class that is unit test and all simple path coverage represents the full coverage of intra class method call sequence as well as integration test. Therefore, the proposed technique supports state, transition and all possible simple path coverage criteria. A state is considered covered if all possible outgoing transitions from the state is traversed at least once. A partially covered state represents a subset of all possible transitions out of the state is exercised. An uncovered state is a state which is never reached through the test suite. This coverage criteria is used by *Coverage Analyzer* component later to identify covered and uncovered elements.

The *Coverage Analyzer* component of this module receives structured UML class, state diagram elements and test suites from *Parser* module as shown in Fig. 2. It then executes the test suite against the state models based on the algorithm shown in Algorithm 1. The algorithm receives existing test suite of the project as input. The test suite contains a number of test scripts. It is assumed that each test script is associated with a class. For each test script, the algorithm identifies the corresponding

Algorithm 1 Coverage Analysis Algorithm

Input: Existing $TestSuite$
Output: Marked state diagram based on coverage

```
1:  Begin
2:  for each TestScript ∈ TestSuite do
3:      get state diagram of TestScript corresponding
            class
4:      for each TestCase ∈ TestScript do
5:          initialize an empty list V to store state variables
             ,class attributes and insert all variables into V
6:          currentState ← GETSTATE(TestStep[0])
7:          for each TestStep ∈ TestCase do
8:              Initialize an empty list T of possible transitions
9:              Initialize boolean variables S, G
10:             Transitions ←
    GETOUTGOINGTRANSITIONS(currentState)
11:             for each t ∈ Transitions do
12:                 S ← MATCHSIGNATURE(t.event, TestStep)
13:                 G ← EXECUTEEXPRESSION(t.guard, V)
14:                 if S AND G then
15:                     insert t into T
16:                 end if
17:             end for
18:             if T = 1 then
19:                 EXECUTEEXPRESSION(t.action, V)
              and mark currentState and t as covered
20:                 currentState ← t.destinationState
21:             else if T >1 then
22:                 Continue with next test case
23:             end if
24:         end for
25:     end for
26: end for
27: End
```

state diagram as shown in line 3 of Algorithm 1. In this step, the algorithm gets the extracted UML class and state elements from the Parser module. The extracted class diagram information of UML contains class name, attribute types, initial value, method name, method parameter name, type and return type. The extracted state diagram elements contains state name, transition, guard condition of each transition (if any) and corresponding action of each transition (if any). The guard condition of a transition is a mathematical boolean expression which decides whether a transition will be taken or not. The action of a transition is also a mathematical operation which occurs after a transition is taken. It is generally used to update value of state variables of the state diagram. A state variable is one of the set of variables that are used to describe the status of the states and changes with the flow of transition.For each test case in the test suite, the state variables and their corresponding initial values are also loaded by the program as in line 5. Once all state variables are loaded, the algorithm then executes the test cases of the test suite.

A test step a is method call statement in the test cases. For the first test step of each test case the corresponding current state is identified in line 6. Then the outgoing transition from the current state is selected as shown in line 10 of Algorithm 1. For each outgoing transition *t* in list *Transitions*, function *MatchSignature* checks whether the test step method call signature matches with the event call signature of transition *t* and *ExecuteExpression* evaluates the boolean guard expression of tansition *t* based on variable list *V* as illustrated in line 12 and 13 respectively. A possible transition *t* from the current state is selected and inserted into list *T*, if its event call signature matches with the test step and its corresponding guard condition evaluates to be true as shown in line 14 of Algorithm 1. If a possible transition is found, the corresponding action of the transition is also executed by the *ExecuteExpression* function as illustrated in line 19 and the current state and transition are marked as covered. Both the guard condition and the action expression are performed by the *Expression Evaluator* component of Figure 2. Line 20 of the algorithm then updates the *currentState* with the corresponding transition destination state for the next iteration. If more than one transition is found, it is discarded and the process continues with the next test case as shown in line 22.

3) Test Regeneration: Test Regeneration module performs the major responsibility of regenerating test cases for the uncovered paths and transitions. There are three components that support the whole test regeneration procedure illustrated in Fig. 2. As mentioned above transitions represent available methods of a class. Therefore, full state and transition coverage ensure high coverage of integration and unit test cases.
The *Unit Test Regenerator* component receives uncovered transitions as input from *Coverage Analysis* component as shown in Fig. 2. If the event call signature of uncovered transitions derived from state diagram, do not match with the actual method syntax of source code, inconsistent and inexecutable test cases may be generated. Therefore, this component also receives the extracted method syntax from *Source Syntax Parser* to ensure consistency. Combining these information it regenerates unit test cases.
The *Integration Test Regenerator* component regenerates integration test cases similar to *Unit Test Regenerator*, it additionally requires the uncovered intra class method call sequences. The *Sequence Identifier* componet supports the *Integration Test Regenerator* by providing the uncovered method call sequences. The *Sequence Identifier* receives the coverage result produced by *Coverage Analyzer* as input. To generate integration tests the uncovered path of the state diagram needs to be extracted. Therefore, this component applies Depth First Search algorithm on the state model to extract possible simple paths. The *Integration Test Regenerator* then regenerates test cases for the extracted paths and these regenerated tests are stored in test scripts to run later.
The *Manual Assert Statement Insertion* component requires human interaction to manually set the assert statements in the test scripts. The *Test Compiler and Test Runner* of Fig. 2 is responsible for setting up required libraries like JUnit for Java programs to run the test scripts. The regenerated test cases can be executed against the state model in the similar way to check the increased coverage. To find out the effectiveness of the proposed technique, an application of its methodology on a sample project is required.

IV. CASE STUDY

The proposed technique is applied on a sample project, named ATM System [7]. The ATM system has some user accounts where the accounts are protected through account number and pin number. If a user enters the right pin number,

the user is allowed to perform some transactions like checking balance, withdrawing and depositing money. A user is allowed to withdraw money if enough money is available in the corresponding user account and in the system cash dispenser. After two failed attempts to insert a pin, the account is locked.

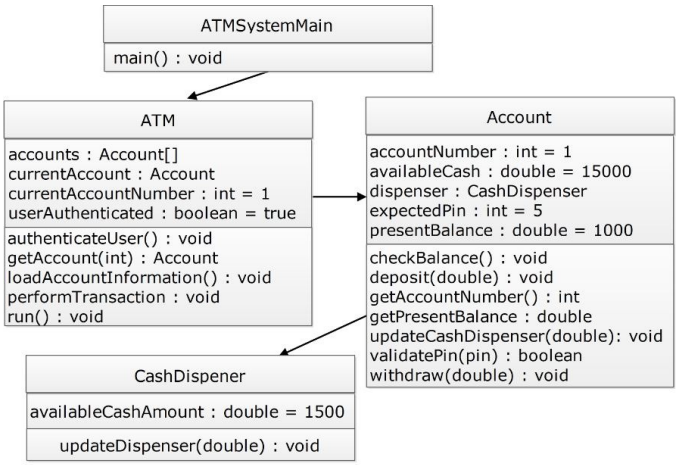

Fig. 3: Class Diagram of Sample Project

The example system contains four classes as shown in Fig. 3, *ATM* which is responsible for running ATM and handling user interaction, *Account* class manages user transactions, *CashDispenser* class updates dispensers available cash and *ATMSystemMain* runs the system. As stated before, each state diagram is associated with a class of the project. The application of the proposed technique on Account class is described below.

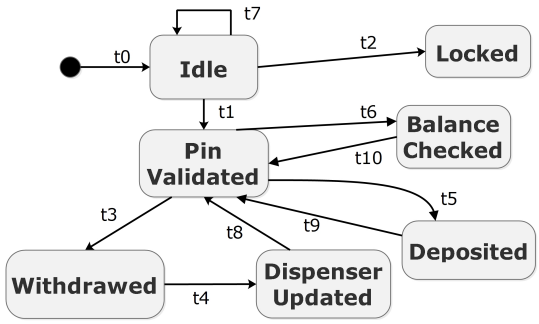

Fig. 4: UML State Diagam (Class: Account)

The state diagram of Account class is shown in Fig. 4. The transitions among the states represent the event call in the class which are illustrated in Table I. The class and state diagrams are then converted to XML using Enterprise Architect [14]. The *UML Elements Identifier* extracts class attributes and methods by analyzing <owedAttribute> and <owedOperation> tags of the converted class diagram XML. On the other hand, information of transitions, guards, actions are also extracted from <transition>, <guard>, <effect> tags of state XMLs respectively.

A snapshot of generated integration test suite of *Account* class by an existing automated test generation framework named SSTF [8] is illustrated in Fig. 5. The test steps of this

```
public class AccountIntegrationTest {
        private Account account;
        @Before
        public void setUp() {
            account = new Account(1,5,1000);
        }
        @Test
        public void SequenceTestWithdraw() {
            account.validatePin(5);
            account.withdraw(200);
            assertEquals(800,account.presentBalance);
            account.updateCashDispenser(200);
            assertEquals(14800,account.availableCash);
        }
        @Test
        public void SequenceTestCheckBalance() {
            account.validatePin(5);
            account.checkBalance();
            assertEquals(1000,account.presentBalance);
        }
```

Fig. 5: Existing Auto Generated Integration Test Suite Example (Class: Account)

TABLE I: Transition Coverage Achieved Through Existing and Regenerated Test Suite (Class: Account)

Transition Id : TransitionEvent [Guard] /Action	**Existing Test Suite [8]**	**Regenerated Test Suite**
t0 : setToIdleState	Covered	Covered
t1 : validatePin(pin) [pin==expectedPin]	Covered	Covered
t2 : validatePin(pin) [pin!=expectedPin AND attemptCount+1>=maxAttempt]	Uncovered	Covered
t3 : withdraw(amount) [amount<presentBalance AND presentBalance-amount>=100 AND availableCash-amount>=100] /presentBalance=presentBalance-amount	Covered	Covered
t4 : updateCashDispenser(amount) / availableCash=availableCash-amount	Covered	Covered
t5 : deposit(amount) / presentBalance=presentBalance+amount	Uncovered	Covered
t6 : checkBalance()	Covered	Covered
t7 : validatePin(pin) [pin!=expectedPin AND attemptCount+1<maxAttempt] /attemptCount=attemptCount+1	Uncovered	Covered
t8 : backToIdleStateAfterWithdraw	Covered	Covered
t9 : backToIdleStateAfterDeposit	Uncovered	Covered
t10 : backToIdleStateAfterCheckBalance	Covered	Covered

generated test suites are then processed by the *Test Step Identifier* component of the *Parser* module described in sub section III-B. For example, Algorithm 1 identifies three test steps validatePin(5), withdraw(200) and updateCashDispenser(200) from the first test case of Fig. 5. The algorithm executes these test steps against the state model of Fig. 4 and evaluates the guard and action conditions based on the method parameters. As a result, Algorithm 1 marks the transition sequence t0 –>t1 –>t3 –>t4 –>t8 as covered. The same process continues for all the test cases and the coverage result is passed to *Test Regeneration Module* for further processing. The class and method syntax of corresponding source files are derived by the *Source Syntax Identifier* of Fig. 2 and also sent to *Test Regeneration* module for consistency checking.

The transition and state coverage achieved from existing test suite of Fig. 5, is represented in Table I and II accordingly. As all the transitions are not covered, therefore, the Account class contains untested methods and intra class method call sequences. These tables clearly show that the test suite failed to cover all the elements of the state model of Fig. 4.

The *Sequence Identifier* component then generates uncovered transition paths based on the coverage result. These generated paths, extracted transition signature and source method syntax are combined, to regenerate executable unit and integration test cases by the *Unit* and *Integration Test Regenerator* accordingly. The generated unit and integration test case sample snapshots are shown in Fig. 6 and 7 respectively. These regenerated test cases are then added to the corresponding existing test suite files. To measure the increased coverage by the regenerated test cases, the *Coverage Analyzer* module again executes the test suites against the state models and computes the coverage result. Table I indicates that 40% transitions are uncovered by the existing test suite generator [8]. Table I and II show that the regenerated unit and integration test suite has successfully covered all the elements of Fig 4.

TABLE II: State Coverage Achieved Through Existing and Regenerated Test Suite (Class: Account)

State Name	Existing Test Suite [8]	Regenerated Test Suite
Idle	Partially Covered	Fully Covered
Pin Validated	Partially Covered	Fully Covered
Withdrawed	Fully Covered	Fully Covered
Dispenser Updated	Fully Covered	Fully Covered
Deposited	Uncovered	Fully Covered
Balance Checked	Fully Covered	Fully Covered
Locked	Uncovered	Fully Covered

```
public class AccountUnitTest {
        private Account account;
        @Before
        public void setUp() {
            account = new Account(1,5,1000);
        }
        @Test
        public void depositeTest() {
            double mockVar0;
            mockVar0 = EasyMock.createMock(double.class);
            account.deposit(mockVar0);
            assertEquals(mockVar0+1000,
                account.presentBalance);
        }
        @Test
        public void validatePinTest() {
            account.validatePin(1);
            assertEquals(false,account.isLocked);
        }
```

Fig. 6: Unit Test Example

```
public class AccountIntegrationTest {
        private Account account;
        @Before
        public void setUp() {
            account = new Account(1,5,1000);
        }
        @Test
        public void SequenceTestDeposit() {
            account.validatePin(5);
            account.deposit(500);
            assertEquals(1500,account.presentBalance);
        }
        @Test
        public void SequenceTestLocked() {
            account.validatePin(6);
            account.validatePin(7);
            assertEquals(true,account.isLocked);
        }
```

Fig. 7: Integration Test Example

V. Conclusion

This paper introduces a test suite regeneration technique to ensure state model coverage, that analyzes the coverage achieved so far, and uses both the UML elements and source syntax for the regeneration. The consideration of coverage analysis result, UML elements and source syntax, while regenerating process achieve high coverage of the state model as well as generate executable unit and integration test cases.

The *Parser*, *Coverage Analysis* and *Test Regeneration* module operate together to regenerate unit and integration test cases. While *Parser* module processes UML, source and test suite information provided by the user, *Coverage Analysis* module identifies the covered and uncovered elements of the state model. *Test Regeneration* module considers the coverage result and extracted information by the *Parser* and regenerates unit and intra class integration test cases.

A case study on a sample project is shown to analyze the effectiveness of the technique. The case study result shows that the existing test suites do not cover all the elements of the state model and full coverage is achieved through the regenerated test cases. The future scope lies in incorporating more coverage criteria with the regeneration process.

References

[1] M. Shahid and S. Ibrahim, "An evaluation of test coverage tools in software testing," in *Proc. of Computer Science and Information Technology and International Conference on Telecommunication Technology and Applications*, vol. 5. IACSIT Press, 2011.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Elsevier Inc., 2007.

[3] E. Najumudheen, R. Mall, and D. Samanta, "A dependence graph-based test coverage analysis technique for object-oriented programs," in *Proc. of the 6th International Conference on Information Technology*. IEEE, Apr 2009, pp. 763–768.

[4] R. Ferreira, J. Faria, and A. Paiva, "Test coverage analysis of uml state machines," in *Proc. of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, Apr. 2010, pp. 284–289.

[5] M. Sarma and R. Mall, "Automatic test case generation from uml models," in *Proc. of 10th International Conference on Information Technology*. IEEE, Dec. 2007, pp. 196 – 201.

[6] W. He and R. Zhao, "Sequential pattern mining based test case regeneration," *Journal of Software*, vol. 8, no. 12, pp. 3105–3113, Dec 2013.

[7] (2015, Aug.) Sample atmsystem project. [Online]. Available: https://github.com/Afrina/ATMSystem

[8] N. Nahar and K. Sakib, "Sstf: A novel automated test generation framework using software semantics and syntax," in *Proc. of 17th International Conference on Computer and Information Technology*. IEEE, Dec. 2014, pp. 69–74.

[9] P. Heckeler, J. Behrend, T. Kropf, J. Ruf, and W. Rosenstiel, "State-based coverage analysis and uml-driven equivalence checking for c++ state machines," in *Proc. of the 2nd International Workshop on Formal Methods and Agile Methods*, Sep. 2010, pp. 49–62.

[10] P. Augsornsri and T. Suwannasart, "An integration testing coverage tool for object-oriented software," in *Proc. of the International Conference on Information Science and Applications*. IEEE, May 2014, pp. 1–5.

[11] N. Alshahwan and M. Harman, "State aware test case regeneration for improving web application test suite coverage and fault detection," in *Proc. of the International Symposium on Software Testing and Analysis*. ACM, Jul 2012, pp. 45–55.

[12] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proc. of 13th European conference on Foundations of software engineering*. ACM SIGSOFT, 2011, pp. 416–419.

[13] M. Chen, X. Qiu, and X. Li, "Automatic test case generation for uml activity diagrams," in *Proc. of the International Workshop on Automation of Software Test*. China: ACM, May 2006, pp. 2–8.

[14] (2015, Aug.) Enterprise architect - uml design tools and uml case tools for software development. [Online]. Available: http://www.sparxsystems.com/products/ea/