International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.7, October 2017

UML Diagrams and Source based Automatic Test Suite Regeneration for Improving State Model Coverage

Afrina Khatun
Institute of Information Technology
University of Dhaka

Naushin Nower
Institute of Information Technology
University of Dhaka

Kazi Sakib
Institute of Information Technology
University of Dhaka

ABSTRACT

Automated test regeneration intends to ensure high coverage of system model from an existing test suite. While regenerating test suite, most of the existing techniques ignore coverage achieved by existing test suite. As a result, these techniques leave important model elements untested. Thus, an automatic test regeneration technique to achieve high state model coverage is proposed. In the proposed technique, Input Parser module processes inputted UML diagrams, source code and test suite as XML elements, source class and test steps respectively. The Coverage Computational module measures model coverage result by executing the existing test suite. Finally, Test Regeneration module regenerates executable test cases considering coverage result, UML and source information. The experimental results on four projects show that the proposed technique improves transition and state coverage of existing test suite on average by 61.26% and 52.95% respectively. Moreover, the technique has also successfully regenerated 98% executable test cases.

Keywords

Software Testing; Automatic Test Regeneration; Model Coverage Analysis; Unit Testing; Integration Testing.

1. INTRODUCTION

Automatic test regeneration refers the generation of test cases to achieve certain goals (such as increased test suite effectiveness, higher test coverage, higher model coverage etc.) using the knowledge of existing test suite. It ensures the quality of a software product by achieving high coverage. As system state diagrams represent software classes, therefore, high state model coverage indicates that all the methods and interactions among the methods of a class are tested. However, existing auto-generated test suite often fails to achieve full coverage of the state model, leaving important model elements untested. As a result, test regeneration needs to be done for covering all elements of the system model. In this context, automatic test regeneration ensuring state model coverage can mitigate the manual efforts for identifying untested elements of the state model, and regenerate test cases for covering these elements in a cost effective way.

Test coverage is a quality assurance metric which indicates how thoroughly a test suite exercises a given system [1]. On the other hand, test case regeneration from existing test suite is required to ensure high test coverage of a System Under Test (SUT).However, the existing coverage analysis techniques conclude their task by only identifying the covered and uncovered elements. These techniques do not regenerate test cases for the untested elements. As a result, the tester needs to manually regenerate test cases for those untested elements. Therefore, an automatic regeneration approach is required to identify the uncovered elements by measuring coverage of the existing test suite, and regenerate executable test cases to cover these untested elements.

As test automation tools lack predefined coverage checking while generating test suites, these tools generally fail to achieve full coverage of the model. Existing coverage analysis tools measure the coverage of a system model, based on some predefined coverage criteria such as state coverage, transition coverage, all path coverage, etc. These tools only identify the tested and untested elements of the system model by the exiting test suite. 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 SUT [2] or source code [3] or both [4]. However, those fail to determine how much coverage is achieved through the generated test suite. As a result, important 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.

Several techniques for test case regeneration, test case genenaration and test coverage analysis have been proposed. Most of the test coverage analysis [5], [6], [7] techniques measure the coverage result, by identifying the covered model elements through existing test suites. These techniques do not generate text cases for the untested model elements. Therefore, the existing test suite fails to test all the elements of the software models. Few test case regeneration techniques [8], [9] have also been proposed in the literature. These techniques regenerate test cases from existing test repositories. However, these techniques ignore already achieved model coverage result of existing test repositories while regenerating test cases. Existing test generation techniques focus only generating test cases from model diagrams or source code or both. Therefore, those techniques fail to cover all the elements of the models, which could lead to manual inspection of whole model.

To overcome the limitations of the above mentioned approaches, the coverage achieved through existing test cases needs to be measured, and this coverage result needs to be analyzed while regenerating test cases. In order to do that, this research incorporates coverage analysis result with test regeneration to generate test cases for the uncovered paths of the model. In order to do that, a technique for automatic test regeneration by analyzing coverage is proposed

to regenerate executable unit and integration test cases for achieving high coverage.
The technique contains three modules to manage the whole regeneration process. The *Input Parser*, *Coverage Computation* and *Test Regeneration* - each module performs specific tasks to support the technique implementation. The *Input Parser* module receives UML diagrams (such as class and state diagram), source code and existing test suite as input. The *Coverage Computation* module uses the processed test suite and UML elements to identify the uncovered elements of the state model by executing the test cases against the model. Finally, the *Test Regeneration* module uses the coverage result to generate all possible uncovered transition paths from the state diagram assuming it as a directed graph. It then regenerates unit and intra class integration test cases for the generated uncovered transition paths. A preliminary concept of the proposed technique has been presented in [10]. However, without rigorous experimental analysis, proper justification of the proposed technique could not be confirmed.

To ensure the applicability and validity of the proposed technique, the previous work [10] has been enhanced by analysing it on four real life experimental projects. The technique has been applied on an existing test automation framework, SSTF [4] for measuring the coverage improvement achieved by the regenerated test suite. To evaluate the technique's competence in terms of executable test regeneration, a new metric named *Executable Test Sequence*, has been introduced in this paper. From the result analysis, it is shown that the proposed technique improves both transitions and state coverage on average 61.26% and 52.95% and also regenerates 98% executable test cases.
The rest of the paper is organized as follows. The existing related approaches in the literature are outlined in 2. Section 3 briefly demonstrates the functionality of the proposed technique. The implementation of the technique and experimental result analysis is presented in Section 4. Finally, Section 5 concludes the paper with some future directions.

2. LITERATURE REVIEW

In the literature, 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 or coverage measurement. Some of the significant works related to this research topic are outlined in the following part.

2.1 Coverage Analysis Approaches

A dependence graph-based test coverage analysis technique for object oriented programs had been proposed by [5] et al. . In that 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. During coverage analysis phase, the graph is marked based on coverage criteria and a coverage analysis report is produced for the tester. Therefore, the approach did not generate test cases for the uncovered part of the graph. 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 for C++ programs has been proposed by Heckeler et al. [6]. This approach links the source code with appropriate states of the transitions and executes existing unit, system and integration tests to identify 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 test coverage. Again combining test regeneration technique with the coverage analysis process could produce more accurate test suites.
Ferreira et al. have proposed a state model based test coverage analysis tool, called MoCAT [7]. The tool uses a UML class, state model, and the test suite as input and generates test cases automatically using Spec Explore based on user defined parameter specifications. It then simulates the execution of the test suite over the model to determine the coverage achieved through the test suite. The MoCAT tool supports transition and state coverage criteria and the coverage result gained through the test suite is represented 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. Therefore, a tester would have to manually prepare test cases for the uncovered elements.
Most of the existing coverage analysis techniques in the literature only focus in identifying tested and untested elements and ignore test regeneration. Thus, these techniques 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.

2.2 Related Approaches

Fraser et al. have proposed a tool called, EvoSuite for automatic test generation of object oriented programs [3]. The approach divides its task into two steps, whole test suite generation and mutation based assertion generation. It implements whole test suite generation as a search based approach. A mutation testing approach is taken for the effective assertion generation and it is implemented as a tool. However, the approach does not consider model requirements and coverage while generating test cases.
An automatic test generation technique to detect operational, use case dependency and scenario faults has been proposed by Sarma et al. [11]. This technique converts the use case and sequence diagrams into Use case Diagram Graph (UDG) and Sequence Diagram Graph (SDG) considering use case actors, start and end states, message invocation etc. It then integrates the UDG and SDG graphs into a system testing graph, and generates test cases based on all use cases and sequence path coverage criteria. The technique also identifies three types of faults: use case initialization faults, use case dependency faults and operational faults. However, the test generation process totally ignores source syntax information and intra class interactions of the available model elements.
Nahar et al. have proposed a framework for automatic test generation using software semantics and source syntax [4]. The technique collects software semantic information by parsing XML formatted UML class, state, and sequence diagrams. It also extracts software syntax information from the source code files. Comparing consistency between semantic and syntax information, it generates unit and inter class integration test cases. However, the technique does not consider the valid sequence of interactions, as well as it also ignores intra class integration test cases.
Mingsong et al. have proposed a technique for automatic test case generation for UML activity diagrams [12]. The approach first randomly generates abundant test cases for a JAVA program under testing. Then, by inserting probes into the member function of the source code and executing test cases, it identifies execution traces. The approach considers activity coverage, transition coverage and simple path coverage criteria. Based on these coverage criteria, it identifies test cases that exercised the activity diagram elements and constructed a reduced set of test cases. Instead of reducing the test

suite, regenerating new test cases ensuring the predefined coverage could make the test suite more accurate.

2.3 Test Case Regeneration Approaches

Few test case regeneration techniques from existing test repositories have been proposed in literature. A sequential pattern mining based test case regeneration technique for object oriented projects is presented by Wei et al. [8]. The approach applies Bi-Directional Extension mining strategy to obtain frequent subsequences of method call from existing test suites. It constructs a set of method subsequences, which occurs more than a calculated threshold value and uses Genetic Algorithm (GA) based approach on these method subsequences to regenerate test cases. The approach does not identify whether the existing test repositories exercised all the elements of the program or not. As a result, the regenerated test cases contains the same but optimized method invocations obtained from the existing test repository. If any method invocation is missing in the existing test repository, the reproduced test cases will fail to cover that.
Alshahwan et al. proposed test regeneration technique for web applications using standard and value based Def-Use (DU) testing [9]. This approach considers HTTP requests from a test suite to form client side requests. It then combines fragment of these client requests to regenerate test cases regarding server-side requests. For constructing a test sequence, this algorithm identifies the HTTP requests that define the state variables or database table name and then identifies the usage of the variables. The main limitation of this technique is that, it fails to regenerate test cases for methods, which are not present in the existing test suite. It is so because it only considers methods contained in the existing test suite. In order to achieve higher coverage, the uncovered method call needs to be considered.

3. AN AUTOMATIC TEST SUITE REGENERATION TECHNIQUE FOR IMPROVING STATE MODEL COVERAGE

In this section, the proposed test regeneration technique is presented. As mentioned earlier, existing test regeneration approaches are unable to cover all the requirements of the software model, because these regenerate test cases by combining existing test cases or using evolutionary approaches. On the other hand, coverage measurement tools conclude by only identifying the tested and untested elements. Thus, the tester needs to manually rewrite the test cases for the untested parts. In either case, the test generation algorithms needs to be changed or test suite needs to be generated from the scratch again. To overcome the aforementioned limitations, an automated test regeneration approach is required which incorporates the coverage result of the existing test suite with the test case regeneration process.

3.1 Overview of the Proposed Test Suite Regeneration Technique

The proposed technique computes model coverage result by using existing test suites and incorporates this result with test regeneration process to ensure high model coverage of the system. The top level overview of the proposed technique is shown in Figure 1. The proposed technique is constructed in such a way that, the coverage result of the state model can be used while test regenerating processes. It consists of three major modules *Input Parser*,

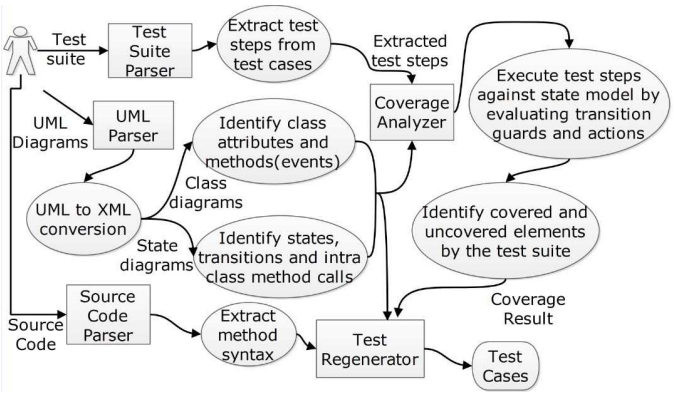

Fig. 1: Top Level View of the Test Suite Regeneration Technique

Coverage Computation and *Test Regeneration* as shown in Figure 1. Each module performs some predefined responsibilities. Input Parser processes the inputted data into a structured form. This module extracts information from the source code, test suite and UML diagrams which are provided by the user. The extracted information is used later by the *Coverage Computation* and *Test Regeneration Module*.
Coverage Computation Module considers each test step of a test case and based on this test step, visits the possible transitions of the state diagrams by evaluating the guard condition and actions. This module then identifies the covered and uncovered elements of the state model based on state, transition and simple path coverage criteria to generate the coverage result.
Test Regeneration Module performs the task of test regeneration. It uses the coverage result, extracted UML elements and method syntax as inputs. Based on the coverage result this module regenerates unit and integration test cases for covering the remaining states, transitions and intra class method call sequences.

3.2 Internal Architecture of the Test Suite Regeneration Technique

The internal architecture of test regeneration technique is shown in Figure 2. Each module consists of some sub components which support its functionality. The detail description of these modules is given below.

3.2.1 Input Parser Module.

For the proposed technique, the *Input Parser Module* acts as input data provider. It receives XML formatted UML diagrams, source code and test suite files from a user defined location. The *Input Parser Module* consists of three sub components – *Source Syntax Identifier*, *Test Steps Identifier* and *UML Element Identifier* as illustrated in Figure 2.
The *UML Element Identifier* sub component receives XML formatted UML class and state diagrams. It then identifies class attribute name, their initial values, method name and method parameter list from class diagrams. As each state machine is usually associated to a class, therefore, class methods represent the signature of transition call events of the state machine. This sub component also extracts information of states, transitions, corresponding guards and action events from the state diagrams. This information is used later by the *Coverage Computation* and *Test Regeneration* module to

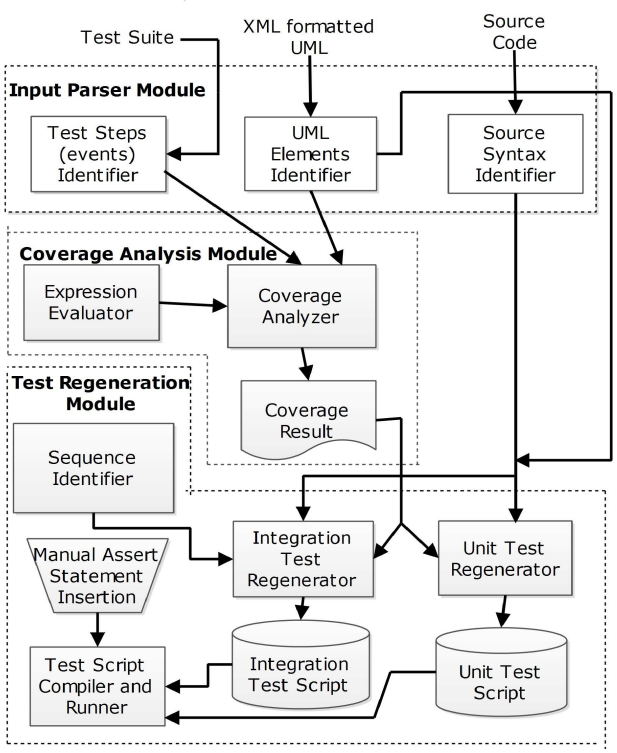

Fig. 2: Internal Modules of the Test Suite Regeneration Technique

compute model coverage result and, regenerate unit and integration tests respectively.

Test Steps Identifier sub component reads existing test suites. It then parses each test case step by step. For a test case, it extracts each method call statement along with its parameter values and passes those to the *Coverage Computation* module. On the other hand, The *Source Syntax Identifier* sub component extracts class object construction, attributes and method syntax from the existing source code files. It then passes those to *Test Regeneration* module for later consistency checks.

3.2.2 *Coverage Computation Module.*

Coverage criteria must be measurable and be an indicator of the test adequacy. *Coverage Computation* module is responsible for the computation of the coverage achieved by existing test suite. One of the coverage criteria applicable to system state models is transition based coverage. State and transition coverage represent the full coverage of all methods of a class that can be verified using unit tests. In addition, all simple path coverage represents the full coverage of intra class method call sequence as well as integration tests. Therefore, the proposed technique supports state, transition and all possible simple path as coverage criteria.

A state is considered fully covered if all possible outgoing transitions from the state is traversed at least once. A partially covered state represents a subset of the outgoing transitions from this state are covered. Moreover, an uncovered state is a state which is never traversed through the test suite. These coverage criteria are used to identify fully, partially covered and uncovered states. This module consists of two components – *Expression Evaluator* and *Coverage Analyzer*.

Algorithm 1 Coverage Analysis Algorithm

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

Begin
for each $TestScript \in TestSuite$ **do**
  get state diagram of $TestScript$ corresponding class
  for each $TestCase \in TestScript$ **do**
    initialize an empty list V to store state variables ,class attributes and insert all variables into V
    $currentState \leftarrow$ GETSTATE($TestStep[0]$)
    for each $TestStep \in TestCase$ **do**
      Initialize an empty list T of possible transitions
      Initialize boolean variables S, G
      $Transitions \leftarrow$ GETOUTGOINGTRANSITIONS($currentState$)
      for each $t \in Transitions$ **do**
        $S \leftarrow$ MATCHSIGNATURE($t.event, TestStep$)
        $G \leftarrow$ EXECUTEEXPRESSION($t.guard, V$)
        if S AND G **then**
          insert t into T
        end if
      end for
      if T = 1 **then**
        EXECUTEEXPRESSION($t.action, V$) and mark $currentState$ and t as covered
        $currentState \leftarrow t.destinationState$
      else if $T >$1 **then**
        Continue with next test case
      end if
    end for
  end for
end for
End

The *Coverage Analyzer* sub component receives structured UML class, state diagram elements and extracted test statements from *Input Parser* module. It then executes the test suite against the state models based on Algorithm 1. The algorithm receives test suite files as input. It is assumed that each class has one intra class integration test script associated with it. For each test script, the algorithm identifies the corresponding state diagram. The extracted state diagram contains state name, state variables, transition, guard conditions (if any) and actions (if any) of transitions. State variables are a set of variables that are used to describe the status of the states and are updated with transitions. The guard condition of a transition is a mathematical boolean expression of state variables which decides whether a transition will be taken or not. The action of a transition is an update operation which occurs after a transition is taken.

For each test case, the class attributes, state variables, and their corresponding initial values are stored in a list called *V* as shown in Algorithm 1. For the first test step of each test case, the corresponding *currentState* is identified in the state diagram. For each current state, an empty list *T* is used to store all possible transitions, and two boolean variables *S,G* are initialized for comparing method signature and satisfying guard conditions respectively. In the next step, the algorithm determines all the outgoing transitions from the current state using the function *GetOutGoingTransitions*. This function takes the *currentState* as a parameter and provides the set of outgoing transitions as output from this *currentStep*.

For each outgoing transition *t* in the list *Transitions*, function

MatchSignature compares whether the test step method call signature matches with the event call signature. *MatchSignature* method considers both method name and parameter list during comparison. Based on this comparison, Algorithm 1 updates a boolean value to variable *S*. *ExecuteExpression* function evaluates the boolean guard expression of tansition *t*. This method evaluates the guard condition based on the value of method parameters and state variables. It returns a boolean value that updates variable *G*. A possible transition *t* from the current state is selected and inserted into list *T*, if above mentioned boolean variables S and G are satisfied.
Both the guard condition and the action expression are handled by the *Expression Evaluator* sub component. If a possible transition is found, the corresponding action of the transition is also executed by the *ExecuteExpression* function. Based on this transition the current state and transition are marked as covered. The algorithm then updates the *currentState* with the corresponding transition's 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. The similar process continues for all the test scripts. Finally, the coverage result is stored in a document to compare with the coverage achieved by the regenerated test cases.

3.2.3 Test Regeneration Module.
Finally, *Test Regeneration* module is responsible for regenerating test cases for the uncovered paths and transitions. There are four components that support the whole test regeneration procedure as illustrated in Figure 2. These sub components are - *Unit Test Regenerator*, *Integration Test Regenerator*, *Sequence Identifier* and *Test Script Compiler* and *Runner*.
After completion of processing, *Coverage Computation Module* passes the computed coverage result to *Unit Test Regenerator* and *Sequence Identifier* sub components for test regeneration. While regenerating test cases, the event call structure of uncovered transitions need to be matched with source syntax to ensure regeneration of executable test cases. As mentioned above, states and transitions represent available methods of a class, thus all state and transition coverage ensure high coverage of unit test cases. Therefore, the *Unit Test Regenerator* sub component receives the uncovered transitions and checks the consistency between source syntax and UML method call signature for regenerating unit test cases.
On the other hand, for integration test cases, possible uncovered path sequences are also need to be extracted. The *Sequence Identifier* sub component supports the *Integration Test Regenerator* by providing those uncovered path sequences. To generate uncovered path sequences, *Sequence Identifier* sub component applies Depth First Search algorithm on the state model. The state machine model of a system is a directed graph. Therefore, all possible simple paths that is paths which do not contain same transition twice and contains uncovered transition are selected as uncovered path sequences and passed to *Integration Test Regenerator* sub component.
The Integration Test Regenerator sub component takes source code syntax and UML information as well as uncovered path sequences as input. Similar to unit test cases, before regenerating test cases, this component compares the event call signature of uncovered transitions with the actual method syntax to avoid inexecutable test cases. It then regenerates integration test cases for the uncovered paths and stores the regenerated test cases in test script.
The *Manual Assert Statement Insertion* sub component requires human interaction to manually set the assert statements and method parameter values by replacing the default values in the test scripts. *Test script Compiler and Runner* is responsible for setting up required libraries (for example 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. Although the proposed approach is implemented in java, the concept of the proposed technique is platform independent.

4. IMPLEMENTATION AND RESULT ANALYSIS

This section demonstrates the effectiveness of the proposed technique by applying it on four sample projects. The effectiveness is measured in terms of number of transition coverage, state coverage and valid executable test sequence generated by the proposed test regeneration approach. This demonstration represents how the proposed technique can mitigate the limitations of the existing techniques which justifies the proposed technique. The coverage achieved by both the existing test suite and regenerated test suite is measured by existing coverage analysis technique MoCAT [7].

4.1 Environmental Setup

The proposed technique was developed using Eclipse Juno Version 4.2 which provides facilities to use JUnit testing framework for creating and running test cases. As the proposed technique evaluates the guard and action conditions of the state transitions, Java Expression Parser library was used for parsing and evaluating those conditions. For checking the method syntax with state diagram method call structure, the source syntax information was parsed using Java Parser. The UML state and class diagrams were converted in XML format using an UML modeling tool, Enterprise Architect. Following is the list of tools used for development and result analysis of the proposed technique:

—Eclipse Juno Version-4.2 [13]
—Java Expression Parser (JEP) [14]
—Java Parser Version-1.0.9 [15]
—Enterprise Architect (EA) [16]
—Eclipse Plugin JUnit 4 [17]

Table 1. : Experimental Projects

Project Name	No. of Classes in Class Diagram	No. of State Diagram	No. of TestScript generated by SSTF [4]
Alarm System	2	1	1
ATM System	4	2	4
Observer Pattern	5	1	1
POAS	23	12	1

The proposed technique regenerates test cases based on the existing test suite and coverage result. Two test case regeneration tools are available in the literature, [9] and [8]. However, these techniques can not be considered for comparative analysis, as the context of test regeneration in these techniques differ from the proposed technique in terms of test coverage criteria. On the other hand, SSTF [4] is an automated test generation technique that considers UML diagrams and source code syntax while generating test cases. Therefore, for measuring the effectiveness of the proposed technique, SSTF [4] generated test suite is used as existing test suite. The effectiveness of the proposed technique is estimated by measuring the extent of coverage improvement, the technique can add to the existing test suite.

4.2 Experimental Projects

For result analysis, the proposed technique is applied on four sample projects which are listed in Table 1 and also available at [18]. The table represents the project name along with the number of classes in the class diagram, the number of available state diagrams, and the number of test scripts in existing test suite, generated using SSTF [4]. All these projects are implemented by considering the actual model requirements, therefore these has been taken as experimental projects for analysis.

"Alarm System" is a simple java project which allows user to set and deactivate alarm for any specific event in a system. The system has a control panel which is connected to one siren and multiple sensors. To activate the system, the user has to press a specific button in the control panel. To deactivate, the user has to insert the right pin. After three failed attempts to insert the right pin, the system becomes locked. If a sensor is activated, the siren starts after few seconds. To shut the siren down, the user needs to insert the correct PIN. This project was used for state based coverage analysis in [7]. From this project, a class diagram containing two classes, one state diagram and one test script (shown in Table 1) generated by SSTF [4] are used as input in the proposed regeneration technique.

"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 also 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. This project was used as a case study in [10]. From this project four classes, two state diagrams and four SSTF [4] generated test scripts are inputted in the proposed regeneration technique.

The "Observer Pattern" is a software design pattern in which a subject notifies its observers about any event by calling one of the methods of the observers [19]. This project was used for automatic test generation in [4] and consists of five classes, one state diagram and one generated test script which are used as input in the proposed technique.

The Program Office Accounting Software (POAS) is an automated accounting software for maintaining the expenditure of running programs (such as bachelor, masters) in educational institutes. For every semester, an amount of money is given to the program office at the beginning of the semester. Based on the given amount and previous semester residual amount, the program committee prepares a budget for that semester.The budget includes five sectors - Office Stationary, Exam Time Refreshment, General Refreshment, Gift and Guest Attendance and Contingency Fund. The source of this project contains 23 classes and 12 state diagrams. Only one integration test script is generated by SSTF for this project, as it ignored the intra class method invocations found in state diagrams.

4.3 Evaluation Metrics

The improvement in the coverage of regenerated test suite has been evaluated by measuring transition coverage, state coverage and the number of executable regenerated test cases.

Transition coverage is the ratio of the number of covered transitions and total transitions in the state diagrams. Achieving full transition coverage for a state diagram refers that all methods of the corresponding source class are called and hence, tested. The ratio is measured using the following equation -

$$TransitionCoverage = \frac{\sum_1^n T_n\,(Covered)}{\sum_1^n T_n\,(Covered) + T_n\,(Uncovered)} \quad (1)$$

where T_n represents the number of transitions in the *n* th inputted state diagram.

State coverage is the ratio of the number of covered states and the number of states in the state diagrams. 100% state coverage of a state diagram depicts that all possible method invocation from the states of the class are tested. State coverage ratio is determined by the following equation -

$$StateCoverage = \frac{\sum_1^n S_n\,(Covered)}{\sum_1^n S_n\,(Covered) + S_n\,(Uncovered)} \quad (2)$$

where S_n represents the number of states in the *n* th inputted state diagram.

The proposed technique checks the source code method syntax while regenerating test cases, as a result executable test cases are generated. Therefore, the effectiveness of the technique is also measured by checking the number of executable regenerated test cases. The result is represented by the ratio of executable test sequences and total generated test sequences. This ratio is calculated using the following equation -

$$ExecutableTestSequence = \frac{Seq_{Executable}}{Seq_{Executable} + Seq_{Inexecutable}} \quad (3)$$

4.4 Result Analysis

The proposed technique is applied on above mentioned sample projects to justify the effectiveness of the regenerated test cases. The number of regenerated integration and unit test cases as well as the number of covered and uncovered transitions for each experimental project by the existing and regenerated test suite are enlisted in Table 2. The table clearly shows that for each project, the number of covered transition by SSTF [4] is low, whereas significant increase in the number of covered transition is achieved by applying the proposed technique. For 'Alarm System', the number of covered transition and uncovered transition are 8 and 9 respectively. However, after applying the proposed technique, 7 integration and 9 unit test cases are newly generated which improves the number of covered transition from 8 to 15.

The proposed technique intends to achieve high transition and state coverage. Therefore, the effectiveness of the technique is represented by the ratio of the number of covered transitions and states

Table 2. : Comparison between the Number of Covered and Uncovered Transitions by SSTF [4] and by Proposed Regenerated Test Suite, and the Number of Regenerated Integration and Unit Test Case

Project Name	No. of Transitions by SSTF [4]		No. of Transitions by Proposed Regenerated Test Suite		No. of Regenerated Test Cases using Proposed Technique	
	Cov.	Uncov.	Cov.	Uncov.	Integration Test Case	Unit Test Case
Alarm System	8	9	15	2	7	9
ATM System	7	9	15	1	3	5
Observer Pattern	2	5	7	0	1	2
POAS	15	70	85	0	23	31

Table 3. : Comparison between Transition Coverage by SSTF [4] and Proposed Regenerated Test Suite Technique and Improvement Ratio

Project Name	Transition Coverage Achieved by SSTF [4]			Transition Coverage Achieved by Proposed Regenerated Test Suite			Improvement Ratio %
	Cov.	Uncov.	Ratio %	Cov.	Uncov.	Ratio %	
Alarm System	8	9	47	15	2	88.25	41.25
ATM System	7	9	43.75	15	1	93.75	50
Observer Pattern	2	5	28.75	7	0	100	71.43
POAS	15	70	17.64	85	0	100	82.36

of the state diagrams. The number of covered and uncovered transitions for each sample project, achieved by existing test suite is also measured by the same existing coverage analysis tool, MoCAT [7]. The ratio of transition coverage result achieved by existing test suite and the proposed regenerated test suite is calculated using Equation 1 and is enlisted in Table 3. In each sample project the ratio of the transition coverage achieved by the proposed technique is twice than the existing coverage ratio. The improvement ratio column in Table 3 represents how much improvement in transition coverage of the existing test suite is achieved by applying the proposed technique. The results clearly depict that on average 61.26% coverage improvement is achieved by the proposed technique.

Table 4. : Comparison between State Coverage by SSTF [4] and Proposed Regenerated Test Suite Technique and Improvement Ratio

Project Name	State Coverage Achieved by SSTF [4]			State Coverage Achieved by Proposed Regenerated Test Suite			Improvement Ratio %
	Cov.	Uncov.	Ratio %	Cov.	Uncov.	Ratio %	
Alarm System	2	4	34	5	1	83.33	49.33
ATM System	8	6	57.14	7	1	87.50	30.36
Observer Pattern	3	3	50.00	6	0	100	50
POAS	12	55	17.91	67	0	100	82.09

Table 4 represents the compariosn of state coverage achieved by SSFT [4] and the proposed techinque. As SSTF [4] ignores intra class interactions while generating test cases, the results demonstrate that only 39.76% average state coverage is achieved using SSTF [4]. On the other hand, average state coverage gained through the proposed regenerated test suite is nearly 92%. The remaining 8% coverage could not be achieved as proper test data was not available in the test method parameters by the testers. However, the improvement ratio represents that the proposed technique successfully improves state coverage of existing test suite on average by 52.95%.
Figure 3 and 4 represents the measure of transition and state coverage respectively of the existing SSFT [4] and proposed regenerated test suite for each sample project. In each case, the proposed technique has been successful to increase the coverage of existing test suite. The bar charts in Figure 3 and 4 show that for each project, the regenerated test suite has higher bar which represents significant improvement in coverage.
Table 5 represents regenerated Executable Test Sequence Ratio,

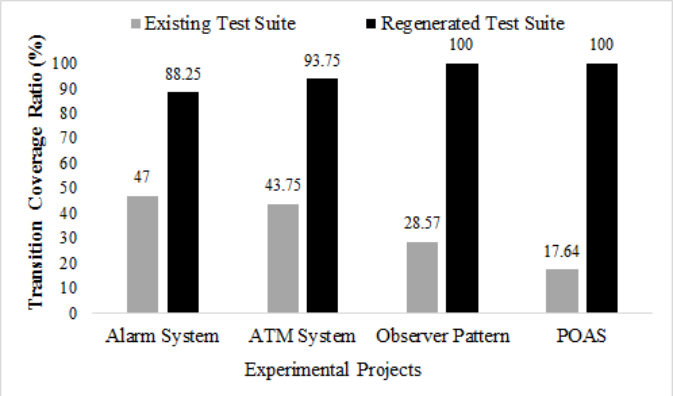

Fig. 3: Comparison of Transition Coverage achieved by Existing [4] and Proposed Regenerated Test Suite

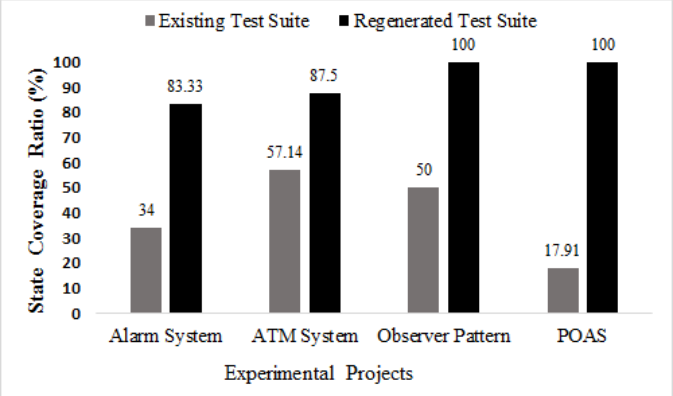

Fig. 4: Comparison of State Coverage achieved by Existing [4] and Proposed Regenerated Test Suite

which is measured by Equation 3. The table depicts that nearly 100% of the regenerated test cases are executable test sequence. However, in the case of 'Alarm System', one inexecutable test sequence is generated, as there is a self-loop transition in that path and the proposed technique considers only simple transition paths. This is because, from state models it is difficult to know the loop breaking constraint in advance for self-loop transitions and hence, leads to different research scope. However, in the rest of the cases almost all the regenerated test cases are valid executable test sequences.

Table 5. : The Number of Regenerated Test Cases by the Proposed Technique and Ratio of Valid Executable Test Cases

Project Name	No of Valid Executable Regenerated Test Cases	No of Invalid Test Sequence	Ratio
Alarm System	15	1	93.75%
ATM System	8	0	100%
Observer Pattern	3	0	100%
POAS	54	0	100%

4.5 Threats to Validity

This section discusses the threats which can affect the validity of the proposed technique. Changes in factors such as implementation

environment, experimental projects, evaluation metrics etc can introduce scopes of improvement for increasing the effectiveness of the proposed technique. Some of such factors are pointed below.

Internal Threats: The internal threats refer threats that affect the validity of the results, depending on the implementation of the technique and the environmental set up of the experimental procedure. The proposed technique as well as the experimental projects are implemented in java programming language. Therefore, the result gained through analyzing the experimental projects can differ when experimented in platforms other than java.

External Threats: The experimental projects that are chosen and the existing test suites used may affect the degree to which the results can be generalized. The experimental projects which are chosen are used in existing techniques (For example, 'Alarm System' was used in [7]). Those projects are also selected as those have associated state models. The existing auto-generated test suite and UML class, state model diagrams which are inputted in the technique can also affect the results gained from the experiments.

Construct Threats: Construct threats are related to the metrics which are used to analyze the effectiveness of the proposed technique. Any change in the metrics may also affect the generalization of the experimental results. The results are analyzed based on transition coverage, state coverage and number of valid regenerated test cases. Therefore, analyzing the results with other metrics can introduce new improvement scopes for the technique.

5. CONCLUSION

Most of the existing coverage analysis techniques conclude their task by only identifying tested and untested elements by the test suite. These approaches do not regenerate test cases for the untested elements. This paper proposes a test case regeneration technique that tends to regenerate test cases to improve the coverage of the existing test suite.

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

The experimental analysis on four projects, depicts that the proposed technique improves the existing test suite on average by 61.26% and 52.95% transition and state coverage respectively. On the other hand, 98% of the regenerated test cases are executable.

The incorporation of sequence diagrams along with the state diagrams can ensure inter class interaction coverage, thus directs to new research scope. The proposed technique measures transition sequence coverage as well as state coverage. Adding more coverage criteria such as branch coverage and guard condition coverage, introduces future tasks. Test data generation techniques can also be incorporated with the proposed technique as a future scope.

6. REFERENCES

[1] Shahid, M., Ibrahim, S. "An evaluation of test coverage tools in software testing." *in Proc. of the 6th* International Conference on Telecommunication Technology and Applications (CSIT), Vol. 5, pp.217–222, 2011.

[2] Utting, M., Legeard, B. "Pratical Model-Based testing A tools Approach", Morgan Kaufmann Publishers, 2007

[3] Fraser, G., Arcuri, A. "Evosuite: automatic test suite generation for object-oriented software", *in Proc. of the 19th* ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering, pp.416–419, ACM, 2011

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

[5] Najumudheen, E. S. F., Mall, R., Samanta, D. "A dependence graph-based test coverage analysis technique for object-oriented programs", *in Proc. of the 6th* International Conference on Information Technology: New Generations (ITNG), pp.763–768, IEEE, 2009

[6] Heckeler, P., Behrend, J., Kropf, T., Ruf, J., Rosenstiel, W., Weiss, R. "State-based coverage analysis and UML-driven equivalence checking for C++ state machines" FM+ AM, Vol. P-179, pp. 49-62, 2010

[7] Ferreira, R. D., Faria, J. P., Paiva, A. C. "Test coverage analysis of UML state machines", *in Proc. of the 3rd* International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), pp.284–289, IEEE, 2010

[8] He, W., Zhao, R. "Sequential Pattern Mining Based Test Case Regeneration", in IEEE Journal of Software, Vol. 8, no. 12, pp.3105–3113, 2013.

[9] Alshahwan, N., Harman, M. "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 (ISSTA), pp.1–5, IEEE, 2012.

[10] Khatun, A., Sakib, K. "An automatic test suite regeneration technique ensuring state model coverage using UML diagrams and source syntax" *in Proc. of the 5th* International Conference on Informatics, Electronics and Vision (ICIEV), pp.88-93, IEEE, 2016.

[11] Sarma, M., Mall, R. "Automatic test case generation from UML models", *in Proc. of the 10th* International Conference on Information Technology,(ICIT), pp.196–201, IEEE, 2007.

[12] Mingsong, C., Xiaokang, Q., Xuandong, L. "Automatic test case generation for UML activity diagrams", *in Proc. of the* International Workshop on Automation of Software Test (AST), pp.2–8, ACM, 2006

[13] Eclipse.org - Juno Simultaneous Release, `https://eclipse.org/juno/`, Online accessed: 19 December 2016

[14] JEP - Java Math Expression Parser `http://www.cse.msu.edu/SENS/Software/jep-2.23/doc/website/index.html` Online accessed: 3 September 2017.

[15] javaparser 1.0.9 - Maven Repository `http://mvnrepository.com/artifact/com.google.code.javaparser/javaparser/1.0.9`", Online accessed: 19 May, 2016.

[16] Enterprise Architect, `http://www.sparxsystems.com/`, Online accessed: 25 August, 2016.

[17] JUnit - About, `http://junit.org/` Online accessed: 18 December, 2016.

[18] Experimental Projects `https://github.com/Afrina/Projects` Online accessed: 5 September, 2016.

[19] Observer Pattern Object Oriented Design `http://www.oodesign.com/observer-pattern.html` Online accessed: 19 December, 2016.