# SOFTWARE SEMANTICS AND SYNTAX AS A TOOL FOR AUTOMATED TEST GENERATION

**NADIA NAHAR**
**BSSE 0327**

A Thesis
Submitted to the Bachelor of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

**BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING**

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

SOFTWARE SEMANTICS AND SYNTAX AS A TOOL FOR AUTOMATED TEST GENERATION

NADIA NAHAR

Approved:

*Signature* *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *Jahanara Akter*, my mother
who has always been there for me and inspired me

# Abstract

Test Automation saves time and cost by digitizing the processes of test generation and execution. Software Requirements Specification (SRS) and source code analysis - these two approaches are generally used for automated test generation, and can be called as semantic and syntactic approach accordingly. Source code parsing in syntactic approach is not enough to identify software semantics and semantic approach lacks syntactic knowledge required for the test syntax creation. To combine these approaches for useful test generation is a research challenge.

A test generation framework is proposed here which uses the information extracted from UMLs and source code. The three layer architecture of the framework is responsible for test generation. The first layer processes the user inputs such as UMLs as XMLs and source code as source classes. The second layer identifies the application semantic from XMLs and extracts syntax from code. It combines the extracted information together and generates unit and integration test scripts. Finally, the last layer is responsible for the execution of the generated scripts.

The comparative analysis of results shows improvement not only in elapsed times but also assures that most of the generated scripts are compilable and runnable. 99.72% of the generated scripts for a sample project set are found to be runnable. Moreover, a case study, conducted on a sample java project, assessed the framework competence and has been successful to construct test scripts. The incorporation of syntax and semantics makes the generated tests less erroneous as it creates a better understanding of the application before the test construction.

# Acknowledgments

I would like to thank Dr. Kazi Muheymin-Us-Sakib for his support and guidance during the thesis compilation. He has been relentless in his efforts to bring the best out of me.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Test Automation is a significant term in software testing. Automation in the field of testing mitigates the manual efforts required for development and execution of test cases. The focus of this research is to decrease the manual effort of test development or generation. This chapter demonstrates the issues of this automated test generation task and introduces the research challenges out of these. It also briefly describes the contribution and achievement of this research. Finally, the organization of this thesis is indicated for giving a reading guideline to the readers.

## 1.1 Introduction

Automated testing is a process where software is tested without any manual input, analysis or evaluation [1]. Automating the testing process may create an impact on software development cost, since testing consumes at least 50% of the total costs involved in development [2]. A tester is considered as a critical analyzer who has profound knowledge on software semantics that is the requirements and the syntax that is the construction details.

Testing is a lengthy process as testers use test plans, cases or scenarios to manually test the software. Automated testing saves time and cost by digitizing this whole process of test generation and execution [3]. However, understanding the program

semantically and incorporating syntax with it without human interaction may either miss some coverage area or produce redundant test cases.

## 1.2 Issues in State-of-the-Art Test Automation

Generally two approaches are used for test suite automation - Software Requirements Specification (SRS) analysis and source code analysis. In SRS analysis, test cases are automatically generated from various UML diagrams - class, state, sequence diagrams and also from GUI screen [4, 5, 6, 7]. This approach can also be referred as semantic approach because the requirements information which is the semantic of software is considered. Another approach of automation testing is the source code analysis which can also be referred as syntactic approach as the software syntactical information is used here for test generation [8, 9, 10]. In this approach, source code parsing is done to identify the class relations and method call sequences. This extracted information is used to form the syntax of the test cases and generates those according to the control flow of the code.

The semantic approach does not always produce effective test cases as it lacks syntactic knowledge which is required for the test syntax creation. Additionally, this approach demands the SRS to be a mirror reflection of the software and assumes the diagrams to be consistent with the code. However, as SRS is created in the early development stage, the diagrams are often backdated and do not match the software code segments completely. On the other hand, parsing done in syntactic approach is not enough to identify the semantic information hidden inside the code. It often results in generation of redundant test suites, and also complete test coverage cannot be achieved. However the lacking of one approach can be compensated by another.

A syntactic test case generation framework was proposed by Pezze et al. [11], where the framework takes the source code and some unit test cases as input. By

extracting method call information it generates complex integration test cases. However, the approach being a syntactic one, fails to completely extract requirements information from inside code, resulting in 40% non-executable test cases. Another syntactic approach is a tool for automatic generation of test cases from source code so as to attain branch coverage in software testing [12]. This tool automates instrumentation process to decrease testing time and errors due to manual instrumentation, and it automatically generates test cases for C/C++ programs. However, the instrumentation time is too high here which could have been improved by considering additional extracted information from UML.

Automatic test case generation considering UML diagrams is an enriched research field. Among the lot, an automatic test case generation approach using UML activity diagrams was presented by Chen et al. [13]. At the same year, Nebut et al. proposed another new approach for the automation in the context of Object-Oriented (OO) software [14]. It used use cases and sequence diagrams for the test generation task. Both these papers used semantic approach for the test generation, and thus suffer from the inherent drawbacks of it as mentioned earlier.

## 1.3 Research Questions

The existing researches on automated test case generation focus either on semantic or on syntactic approach. The shortcomings of these individual approaches have been discussed above. An integration of these approaches is needed to overcome the drawbacks. Thus, this leads to the primary research question -

- How can the semantic and syntactic approach be incorporated together to generate valid test cases?

In a nutshell, the research will incorporate both the semantics and syntax of software to generate unit and integration test cases automatically. A framework is needed here that will use the information extracted from UML diagrams and

source code for the test generation. This can be achieved through answering following sub-questions -

1. How to develop a framework incorporating semantic and syntactic information to automatically generate test cases? What will be the structure of the framework?

Semantic information needs to be extracted from UML models, and control flow information and software syntax need to be identified from source code. Then the gathered information must be incorporated together to generate test cases. This process can be implemented as a framework which will take source and UML diagrams of software as input and produce test cases of the software as output.

1. How to measure the effectiveness of the framework? What are the performance metrics?

After the implementation of framework, it can be compared with the existing ones for the measurement of effectiveness. The elapsed time, number of generated test cases, executable test case ratio etc. can be used as metrics for the performance measurement of the generated test cases.
Answering these questions and providing a solution for automatic test generation, with incorporation of software syntax-semantics, are the objectives of this research.

## 1.4 Contribution and Achievement

This research incorporates both the semantics and syntax of software to generate unit and integration test cases automatically. A test generation framework is proposed here which uses the information extracted from UML diagrams and source code. It works in three layers based on the three categories of tasks it needs to manage such as input processing, test generation and test execution. The User End, Service and Test Run Layer - each have some predefined responsibilities to carry

on. The User End Layer consists of UML Reader, XML Converter and Source Reader which processes the user inputs such as UML diagrams and application source code. The Service Layer is responsible for the generation of test cases. It receives the processed data from User End Layer and does further computational operations to construct test scripts. Six major and two supporting components work together for extraction of application syntax, semantic; and their incorporation in test generation. This incorporation is done by combining the extracted method call sequence information from UMLs, with the object initialization and method call syntax retrieved from source. Finally, the Test Run Layer has the role to insert assert statements, compile and run tests.

The proposed framework has been implemented using the programming language, java. The evaluation of its competence has been done by applying the framework on four sample projects and analysing the results. These projects of the sample set are also written in java. For the comparative analysis, an existing state-of-the-art approach, Fusion [11] is also applied on the same sample set. The results of both these tools are compared on the basis of execution time, ratio of execution time per generated test suite and percentage of runnable test suites. The analysis shows improvement not only in elapsed times but also assures that most of the generated scripts of the proposed framework are compilable and runnable.

A case study has been conducted here for the assessment of the proposed approach. The case study is carried out on a simple java project illustrating a popular design pattern, the observer pattern. The sample project source code contains an observer class – Person class, a subject class – Product class and the Observer-Subject interfaces. The corresponding UMLs of the sample projects are also built. These source and UMLs are inputted in the User End Layer. The output are some XML data and source classes. These are received by the Service Layer and after processing, the output is the test scripts. These scripts are then successfully run by the Test Run Center.

## 1.5 Organization of the Thesis

This section gives an overview of the remaining chapters of this thesis. The chapters are organised as follows –

**Chapter 2:** Some preliminaries of software testing are discussed along with the basic concept of automated testing.

**Chapter 3:** To the best of author's knowledge, no existing literature incorporates software source code with software UMLs to generated test scripts. This chapter shows the existing researches in test generation automation.

**Chapter 4:** The architecture of the proposed framework is briefly demonstrated in this chapter.

**Chapter 5:** The implementation of the framework and comparative result analysis is presented here.

**Chapter 6:** A case study on a sample java project is shown here for the assessment of the proposed approach.

**Chapter 7:** It is the concluding chapter which contains a discussion about the framework and some future directions.

# Chapter 2

# Background Study

Software testing is an important phase in Software Development Life Cycle (SDLC). However, the testing term was introduced only three decades ago. It was renowned as 'Debugging' before that. The separation of debugging from testing was introduced by Myers et al. in 1979 [15]. This chapter shows the evolution of software testing along with different levels of testing. Moreover, automated testing is also briefly analysed here.

## 2.1 Evolution of Software Testing

*"a successful test is one that finds a bug"* [15, 16] — Glenford J. Myers in 1979

*"More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest."* [17]

— Boris Beizer in 2003

With flows of time, software testing pursued different goals and went through different phases. The stages of testing growth are [18] -

- until 1956 - Debugging oriented
- 1957-1978 - Demonstration oriented
- 1979-1982 - Destruction oriented
- 1983-1987 - Evaluation oriented
- 1988-present- Prevention oriented

## Debugging Oriented Stage

In this phase, testing was correlated with hardware only. Although, software bugs were still present, those were considered to be a part of hardware testing. There was no fine line between "debugging" and "testing". Different groups considered those differently, but no one could describe the distinctions between those. Both of those were considered to be sub-activities of "check out". Alan Turing wrote the earliest article on program checkout in 1949 [19].

## Demonstration Oriented Stage

In this stage of software test, Charles Baker differentiated test from debug. In a review [20] of Daniel D. McCracken's book 'Digital Computer Programming'[1], he identified program checkout to have two goals: "Make sure the program runs" (debugging) and "Make sure the program solves the problem" (testing). Thus, the meaning of those terms were the same as per attempt to detect and correct bugs but were different as per definition of success.

## Destruction Oriented Stage

Myers first defined testing in 1979 which triggered the initiation of this stage - *"the process of executing a program with the intent of finding error"* — G. J. Myers

[1]D. D. McCracken, Digital computer programming. John Wiley & Sons, 1957

The core concern of this phase was to detect fault which obviously led to the shifting of spotlight from demonstration to detection. The fault detection approaches of this stage can be found in articles of Deutsch [21] and Howden [22].

### Evaluation Oriented Stage

A methodology for product evaluation during SDLC, targeted at Federal Information Processing System (FIPS) was provided in this stage. In 1983, a guideline was published by the Institute of Computer Sciences and Technology of National Bureau of Standards for this purpose [23]. The basic, the comprehensive and the critical - these three techniques were recommended by the guideline. While the destruction stage only detected implementation faults, this stage focus to detect requirements, design and implementation faults.

### Prevention Oriented Stage

This is the current stage of testing. This is the first stage to describe prevention of faults along with detection of faults in evaluation stage. This stage includes certain review and analysis activities in testing. It supports creation of test plan, designing of test and evaluation of test. This test analysis is used for improvement of requirements specification as well as design; and the test execution assures improvement of coding.

## 2.2 Levels of Testing

Four most recognised testing levels are: unit, integration, system and acceptance testing. However, recently component level testing has also gained popularity and now considered as a level of testing. These different levels of testing are demonstrated in this section.

## Unit Testing

*"A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A unit is a method or function."* [24] — Roy Osherove

Kent Beck first introduced unit testing for Smalltalk. He developed a testing framework package for Smalltalk, which was later named as SUnit (SmalltalkUnit) [25]. Unit tests are usually written by developers and is a type of white-box testing (based on code). This type of testing is done to assure that a specific function is working as expected. In simple language, it can be said as -

*"Unit testing is testing of individual units or groups of related units"* [26]

— Per Runeson

## Integration Testing

Unlike unit testing, integration testing seeks interface between components or modules of software. It can be based on functional decomposition, call-graph or paths. There are various strategies of integration testing – top-down, bottom-up, big bang etc. Dependency of program segments are the main reason for the evolution of integration testing. 40% of software errors can be traced to integration problems which emphasises the importance of integration testing [27].

## Component Level Testing

Component Based Software Engineering (CBSE) emphasised the importance of components in software development. And so, component level testing has gained popularity nowadays. Although, this type of testing was known from early stages of testing, recently it has been considered to be the primary technology to overcome the software crisis [28, 29]. The failure of ARIANE 5 rocket on June 4th, 1996,

brought the spotlight on the importance of component level testing [30]. Basically, data-pass between various units, or subsystem components are tested by this type of testing unlike full integration testing.

### System Testing

A complete integrated system is tested by system testing to verify if the system meets its requirements. In simple words, the System Under Test (SUT) is compared to match its intended specification [31]. It is performed in context of Functional Requirement Specification(s) (FRS) or a System Requirement Specification (SRS) [32]. Not only the design, but also the system behaviour, customer expectation are considered in this level of testing.

### Acceptance Testing

Acceptance testing focuses on the fact that if the requirements specification and the contract are met by the product.

*"Acceptance testing (AT) is a testing process that is applied to ascertain whether the software products under development have met the requirements specified by agreement, e.g., contract."*[33] — J. John Marciniak

Alpha and Beta testing are common types of acceptance testing. Alpha testing takes place in developer's place while beta testing is performed in customer's place.

## 2.3 Automated Testing

Ever-shrinking schedules and minimal resources are the daily issues, todays software managers and developers deal with [3]. More than 90% of developers miss ship dates. For 67% developers, missing deadline is routine occurrences. Moreover, 91% remove key functionality for meeting deadline [34]. In this circumstance, software testing is an extra headache having a huge requirement of time and cost.

Along with consumption of time, testing consumes at least 50% of the total costs involved in development [2]. Hence comes the significance of test automation. In test automation, software is tested without any manual interaction [1]. This saves both development time, and cost required for the management of extra human resources. Automated testing can be defined as -

*"The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool*[3] — Elfriede Dustin

However, in spite of the significance, only a small percentage of tests are actually automated today [35]. The gap between formalized research methodologies and industrial practice needs to be mitigated for making test automation useful.

## 2.4 Summary

A discussion of the evolution of software testing is done in this chapter. Software testing started its journey on Debugging oriented phase. And now, after crossing three more stages, it has reached to a matured phase, the Prevention oriented stage. The levels of testing are also more developed now and are introduced one-by-one in this chapter for a better understanding of software testing. Finally, a significant discussion of test automation has taken place. In the following chapter, some automated test generation literatures are presentated.

# Chapter 3

# Literature Review of Automated Test Generation

In the literature, several automatic test case generation techniques have been proposed. Most of those techniques consider either semantic or syntactic approach. As stated previously, SRS analysis and source code analysis - these two approaches can be termed as semantic and syntactic approach accordingly. In SRS analysis, which deals with the software semantics, various UMLs - class, state, sequence diagrams as well as GUI screens are analyzed. Similarly, source code analysis or the so called syntactic approach uses the software syntax for the test generation. Some authors have also focused on regression test generation and some have brought other concepts like user interaction, decision table for the generation of unit test cases. In this chapter, these literature contributions will be discussed.

## 3.1 Semantic Approach

The literature of test generation automation by semantic approach is a rich field. Nebut et al. proposed a new approach for the automation in the context of object oriented (OO) software [14]. A complete and automated chain for test cases was

derived from formalized requirements [36] of embedded OO software here. The proposed approach generates system test scenarios from use cases. The approach is based on use case models unraveling the ambiguities of the requirements written in natural language. At first, relevant paths named as test objectives were extracted from a simulation model of the use cases using coverage criteria. Generation of test scenarios from these test objectives were done next. Automation of the test scenarios generation was done by replacing each use case with a sequence diagram. However, only the sequence diagrams were considered in the generation process, while it might have been more interesting to use activity and state diagrams in certain circumstances.

An automatic test case generation approach using UML activity diagrams was presented by Mingsong et al. [13]. In spite of using activity diagrams directly to generate test cases, an indirect approach to select tests from the set of the randomly generated tests were presented here. Randomly generated tests were first executed and corresponding program execution traces were got from it. These traces were compared with the activity diagrams according to some test adequacy criteria such as activity coverage, transition coverage, simple path coverage. The abundant test cases were then pruned out and the reduced test set were provided which met the criteria. It can also check the consistency between specifications and corresponding programs. This approach corresponds to Model Driven Architecture (MDA) [37] and the Model Driven Testing [38]. However, the generation of basic test cases in the paper might have been led to generation of complex test cases by considering the other UML diagrams such as state diagram, sequence diagram; and at the same time, the source information could also been put in good use here.

Recently, model based testing has gained popularity [39] and researchers are exploring software models usage to support verification and validation (V&V) activities [40]. Javed et al. established a method that uses the model transformation

technology of MDA to generate unit test cases by using sequence diagrams [6]. The generation task is done in two levels. Firstly, test cases are generated from sequence of method calls in sequence diagrams and are selected by the tester. Then the test results are checked by comparing expected and actual return values of the selected method calls, and by comparing the execution traces with the calls in the sequence diagram. The method was implemented using Eclipse, Tefkat, MOFScript, JUnit and SUnit; and is general for implementation in any other xUnit testing framework. However, UML diagrams are designed in the early stage of software development process so as the sequence diagrams. Later on the software under development may have changed requirements. Moreover, as the agile development process is taking over other software development models nowadays, the requirement and feature change has become a common phenomenon. In this situation, the UML diagrams cannot be fully relied on. The generated test cases from the UMLs may become outdated for the software test. The paper does not consider this fact.

Another method for model-based test generation is described by Enoiu et al. in [7]. It is for the safety-critical embedded applications implemented in a programming language such as Function Block Diagram (FBD) [41]. The approach is based on functional and timing behavior models and a model checker is used by it to automate generation of test suites. The automatic generation of unit test suites from a formal model along with specific coverage measurements is contributed here. Firstly, transformation of FBD programs into timed automata models was done. Then UPPAAL [42] model-checker was used for performing symbolic reachability analysis on those. The generated diagnostic trace, witnessing a submitted test property or coverage criteria, was later used to produce test suites. Finally, The applicability of the method was demonstrated on a real world train control and management software. However, in addition to unit testing, test generation to support integration and system level testing could also to been considered.

## 3.2 Syntactic Approach

A prominent literature of the syntactic approach proposed a tool for automatic generation of test cases from source code [12]. This tool allows generating branch coverage test cases for programs written in C/C++. It has several modules:

**Parser** generates the control flow graph of the source code

**Instrumenter** generates the instrumented source code

**Test Cases Generator** generates the test cases, using the instrumented source code and the control flow graph

The tool implemented a Tabu Search based algorithm [43], a Scatter Search based algorithm [44] and a Random algorithm as the Test Case Generators. The testing time and errors due to manual instrumentation have been decreased by the use of this automation tool. However, the tool consumes more time than expected as it needs to instrument the application source code first. As per the result shown, the instrumentation of an example code needs 30 minutes of time while the test run time is in seconds. This vast amount of time consumption can be easily avoided by incorporating UML information with the source code.

Fix et al. presented the design and implementation of a framework for automated unit test code generation [10]. A process for semi-automatic generation of unit test code was described here along with the framework, developed in .Net managed environment, using eXtensible Stylesheet Language Transformation (XSLT), eXtensible Markup Language (XML) and the C# programming language. The framework starts with a program that extracts method information from the system under test (SUT). This information includes the method signatures - method name, input and output parameters, and return type. An XML metadata file is produced according to the information. An XSLT transformation stylesheet file is then manually created based on the XML metadata file. This XSLT file contains

the logic that generates the unit tests for the methods of the SUT. The framework used a very simple generator program which accepts the XML metadata file and the XSLT transformation instructions as inputs, to produce the unit tests. A software test engineer loads the generated unit tests into a test run environment, such as the Visual Studio IDE or NUnit runner program, and executes the unit tests. However, the generation of basic test cases here could have been easily improved by incorporation of semantic information and it could also lead to generation of complex test cases which was not considered in this paper.

A new approach to use aspect-oriented programming (AOP) [45] for testing was proposed by Duclos et al. in [9]. An automated aspect creator named ACRE was presented here to perform memory, invariant and interferences testing for software programs written in C++. Developers, having no knowledge of AOP, can use ACRE aspects to test their programs without modifying the behavior of the source code. ACRE uses a domain-specific language (DSL) [46], which are statements that testers insert into the source code like comments to describe the aspects to be used. The presence of DSL statements in the source code does not modify the programs compilation and behavior. ACRE parses the DSL statements and automatically generates appropriate aspects that are then weaved into the source code to identify bugs due to memory leaks, incorrect algorithm implementation, or interference among threads. However, there are several validity threats of ACRE. An internal validity threat is that the founded bugs may be caused by the aspects added to the source code, which means the tool to detect bug in a program can itself create bug in it. An external validity threat is that ACRE has been used only under Linux. And also that, ACRE only being applied on NOMAD, a large C++ program used in industry and research, does not prove it to work as expected with other programs.

An approach to generate complex or integration test cases using simple or unit ones has been proposed by Pezze et al. [11]. The key observation of the paper was

that unit and integration, both test cases are produced from method calls which work as the atom of those. Unit tests contain detailed information about class instantiation, method call arguments construction and other application syntactic knowledge on individual state transformations caused by single method calls. This information which was used to construct more complex integration test cases focusing on the interaction of the classes. The main contribution of the approach is the idea of combining initialization and execution sequences of unit test cases to generate integration test cases. The approach is effective in the cases when unit test scripts are already present. However, the generation of unit test scripts is not in the scope of this paper. Moreover, the approach used for the test generation is syntactic approach and fails to completely extract requirements information from inside code. This leads to the generation of redundant test scripts which cannot be compiled and run. About 60% of the generated test cases compile and execute immediately and the leftover 40% are non-executable. If the semantic information would have been incorporated with the extracted source information, the application behavior could be clearer and the result could have been improved.

## 3.3 Other Approaches

Regression test generation has also emphasized by some authors. Taneja et al. presented an automated regression unit test generator called DiffGen for Java programs [47]. Differences between two versions of a given Java class were evaluated by checking observable outputs and receiver object states first.The detected behavioral difference provoked regression test generation next. Change Detector, Instrumenter, Test Generator and Test Execution components worked together for this test generation task. However, the limitation of DiffGen prevents it to detect changes on fields and methods or signatures. If UML diagrams were considered and compared along with the source, this limitation could have been resolved.

Moreover, a black-box testing technique was considered by Sharma et al. to test suite auto-generation [48]. A generic novel framework was proposed here which processes decision table for the generation and eliminates the redundancy in test case. The major focus of the paper was on unit testing which could be extended to integration testing, only if some basic UMLs along with the decision table be considered for information extraction.

## 3.4 Summary

As stated above, these research address the importance of test generation automation. Although various automation frameworks have been proposed throughout the years, none of those can be identified to be complete and flawless. Researchers have applied different approaches for improving the result of automation, but complete success could not be attained yet. Incorporation of these approaches can encompass the desired objective, so further research is needed here. The contribution of this research will be discussed in the next chapter.

# Chapter 4

# SSTF: A Novel Test Automation Framework

A new automated test generation framework is proposed in this chapter. As stated in previous chapters, the syntax of software is the set of rules that defines the combinations of symbols, considered in that software language and can be identified by parsing the software source code. On the other hand, software semantics is the field concerned with the meaning of software languages which can be recognized by analyzing software UML diagrams. Test cases can be generated from source code that is on the basis of syntax; or from UML diagrams that is on the basis of semantics. Using only one of this information is not enough to generate effective test scripts as it cannot extract the software information faultlessly. Studying existing frameworks revealed that those do not consider both syntax and semantics of software together. Thus a new framework is required to support this test generation paradigm. Keeping the above factors in mind, a new automated test generation framework named Semantics & Syntax Test generation automation Framework (SSTF) is proposed.

## 4.1 Overview of SSTF

During the design of the framework, attention is given on the syntactic as well as the semantic knowledge as a combination of these is needed to understand the software as a whole. Source code is used for the collection of syntax while the semantic is assembled from the UML diagrams of the software.

The top-level overview of the proposed architecture is shown in Figure 4.1. The

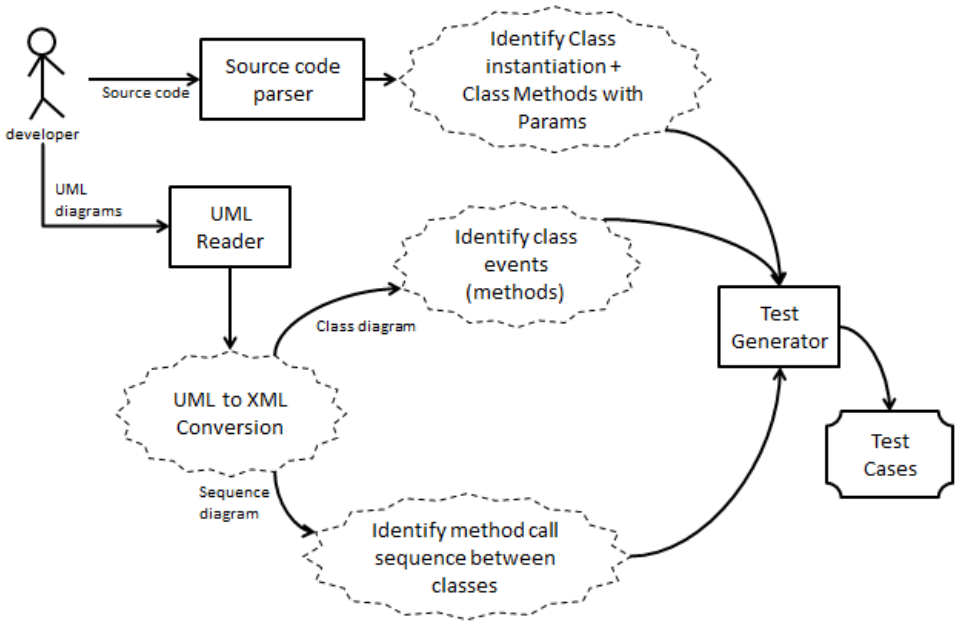



Figure 4.1: Top Level View of SSTF

architecture is divided into three sections as the whole framework stands on three core tasks. The sections are:

- Source Code Parser
- UML Reader
- Test Generator

The first section is the Source Code Parser which is assimilated with the IDE of the clients. The parser is designed by highlighting on the fact that it will identify the software syntax by classifying object construction and method call structure

with parameters. The next component is UML Reader that works as the semantic identifier. The provided UML diagrams of the software, the class and sequence diagrams, are first converted to program readable format (for example, XML) and then read to recognize the software semantics. The final component is a Test Generator that will merge the knowledge gathered from source and UML; and unite those in test cases. Both the unit and integration test cases are produced here, with the help of information taken from class and sequence diagrams accordingly.

## 4.2 Architecture of SSTF

The architecture of the proposed framework is presented in Figure 4.2. The component stack of the architecture can be represented in two categories. The thick bordered components are proposed in this research while the thin bordered are the supporting components to those.
The framework is separated in three layers:

- User End Layer
- Service Layer
- Test Run Center

Each layer has different responsibilities. The User End Layer is the door, through which the users interact with the framework. It consists of three components - UML Reader, XML Converter and Source Reader. The second layer of the framework is the Service Layer. It is the main layer as all the major computations are done here. There are six major components in it and two more supporting components for associating those. The last layer is the Test Run Center. This layer is responsible for running the generated test cases. It has three components for performing this task; Manual Assert Statement Inserter or Test Editor, Test Script Compiler and Test Script Runner.

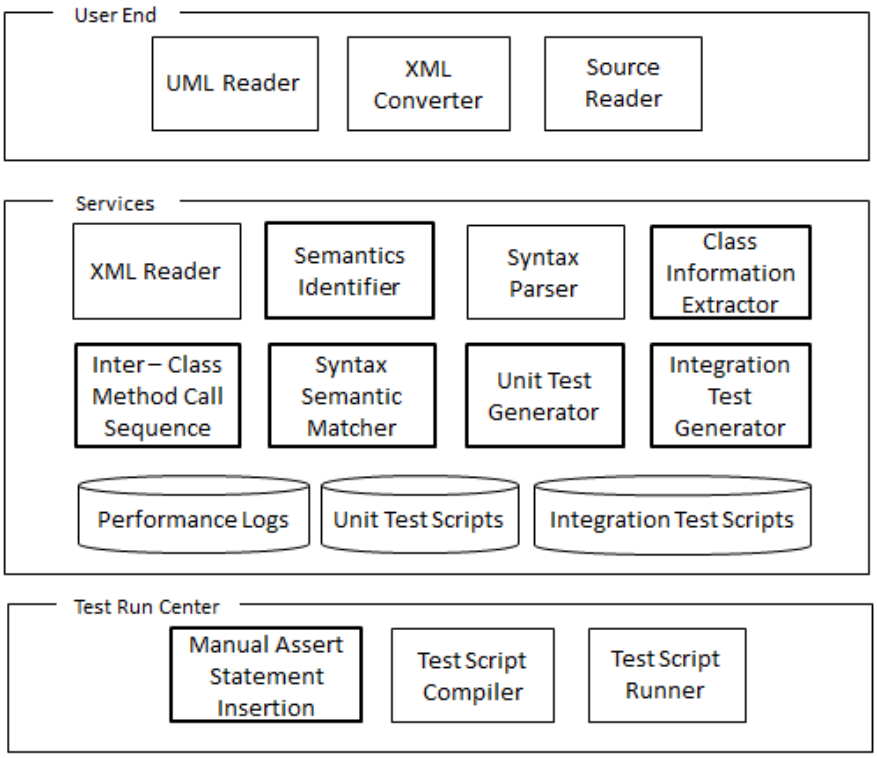



Figure 4.2: The Component Stack of SSTF

As mentioned earlier, the first layer is composed of three User End components. The UML Reader takes UML diagrams provided by the developer and forwards those to the XML Converter component. Usually a computer program cannot perceive information from a UML diagram; this is because program cannot take any input directly from the diagrams. This leads us to the conversion of the diagrams to a program readable format such as XML. The XML Converter component is introduced for this purpose. It produces XML files that can be later read by the framework. An example of proposed XML file structure is shown in Figure 4.3. The third component of this layer is the Source Reader which takes the location of the project source and reads the source files. Figure 4.4 shows interaction among the components of this layer.

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
    <xmi:Extension extender="Enterprise Architect" extenderID="6.5">
        <elements>
            <element xmi:idref="EAID_25DB71D2_C4DF_412a_8488_0038F9C1FB3A" xmi:type="uml:Class" name="Person" scope="public">
                <attributes>
                    <attribute xmi:idref="EAID_8A76C062_7F0F_4eed_B684_D6A3338BAE92" name="personName" scope="Private">
                    </attribute>
                </attributes>
                <operations>
                    <operation xmi:idref="EAID_004CF887_09D2_4cdc_A631_8607E65C0CE6" name="getPersonName" scope="Public">
                    </operation>
                    <operation xmi:idref="EAID_787223EC_1888_443c_ACD3_E36EE5274CB2" name="setPersonName" scope="Public">
                        <parameters>
                            <parameter xmi:idref="EAID_RETURNID_1888_443c_ACD3_E36EE5274CB2" visibility="public">
                                <properties pos="0" type="void" const="false" ea_guid="{RETURNID-1888-443c-ACD3-E36EE5274CB2}"/>
                            </parameter>
                        </parameters>
                        <xrefs/>
                    </operation>
                </operations>
            </element>
        </elements>
    </xmi:Extension>
</xmi:XMI>
```

Figure 4.3: The Proposed Structure of XML Produced from UML Diagram using Enterprise Architect

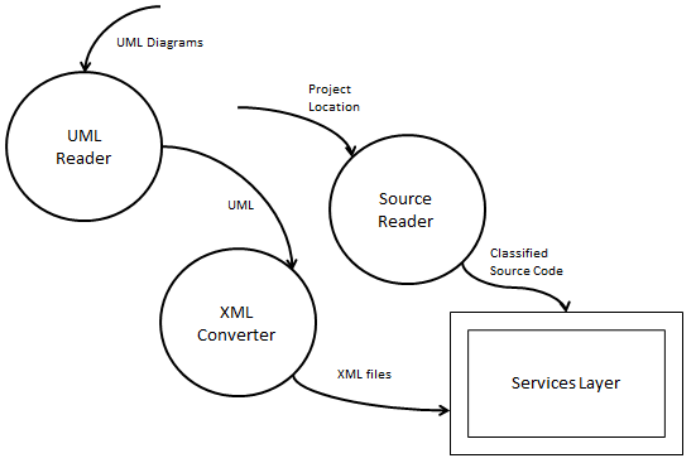



Figure 4.4: The Interaction within the Three Components of the First Layer

In short, the Source Reader takes project source location as input and reads the java files to be processed later by Syntax Parser in second layer. The Project UML Diagrams are inputted in UML Reader next. These are then converted to XML using the XML Converter Component.

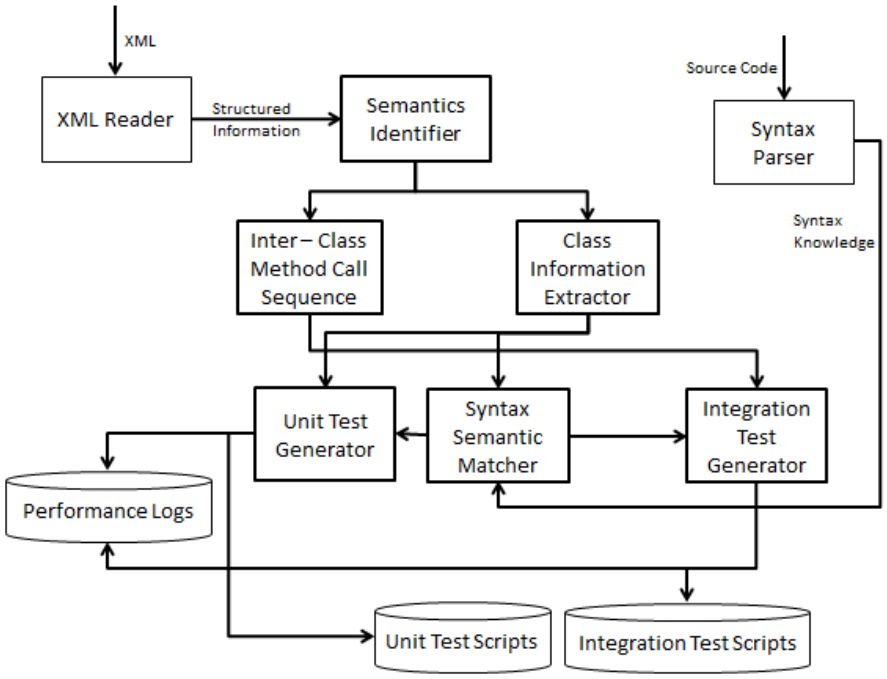



Figure 4.5: The Interaction among the Components of the Second Layer

The six major components of the second layer are mostly responsible for the generation of test cases. The components are the Semantics Identifier, Class Information Extractor, Inter-Class Method Call Sequence, Syntax Semantic Matcher, Unit Test Generator, and Integration Test Generator. Figure 4.5 illustrates the interactions within those that follow after receiving information required from the first layer. The two supporting components of this layer are the XML Reader and Syntax Parser. The XML Reader Component receives files from the XML Converter Component and reads the information hidden there. The Syntax Parser gets the source code from the Source Reader and takes out the syntactic information of the software. Finally information produced by these two components is used by the mentioned major components of the layer.

The Class Information Extractor and Inter-Class Method Call Sequence can be entitled as helpers of the Semantics Identifier. These helpers support it for the identification of the software semantics by categorizing the tasks to be done. The Class Information Extractor pulls out the information of each class in the class

diagram XML. It not only identifies the methods of the class, but also stores the variables, class responsibilities and all other small details such as class association, class role etc. mentioned in the XML. The Inter-Class Method Call Sequence Component is in fact the sequence diagram information extractor. Sequence diagram contains the class to class interaction through method calls. These interaction sequences are identified as method call sequences among classes and can be extracted by this component. These class interactions are later used to generate integration test cases.

The Syntax Semantic Matcher is another significant component of this framework. The main purpose of this component is to identify if the given syntactic and semantic information match or not. Mismatch can happen because of backdated UMLs or incomplete source code. The Matcher Component identifies these conflicts between this two information sources and minimizes its effect on test generation.

The Unit Test Generator and Integration Test Generator are the most important components of this framework. These components use the information produced so far by the other components and generate useful test cases by analyzing those. For generating unit tests the class method information is enough. These information contains the required method name, number of method parameters, method parameters type etc. On the other hand, integration test needs multi-class method call sequence. These sequence is already extracted from sequence diagrams by Inter-Class Method Call Sequence component. Thus the method information along with call sequence can lead to successful generation of integration test scripts.

The Unit Test Generator takes input as follows: class information extracted from the class diagrams and syntax information from the source code. These information are refined by the Syntax Semantic Matcher first. The matched methods of the matched classes are selected for the unit test generation. The syntax infor-

mation provides the syntax of these selected methods. The parametrized method calls from the syntax are afterwards incorporated to generate the unit test scripts; and are stored to be compiled and run later to test the software processes.

The Integration Test Generator uses the class interaction information hidden in the sequence diagrams along with the syntax information and generates integration test scripts. The method calls in between classes are concealed in the sequence diagrams, is revealed by the Inter-Class Method Call Sequence Component, this method call information works as input in the generation of integration tests. Finally both the unit and integration test scripts are stored to be run later by the user. The performance measurements are also stored throughout this whole process to be analyzed later.

The last layer is in charge of running the test cases generated by the second layer. The Manual Assert Statement Insertion Component of the layer needs human interaction for the supplement of Assert Statements inside the generated test scripts. The Test Compiler and Test Runner work together to run the test cases and detect the software faults (e.g. JUnit for Java projects).

## 4.3 Summary

The architecture of SSTF is discussed in this chapter. The core components of SSTF are explained one by one. Finally, a basic set of interactions in order to generate the test cases is shown with example. The incorporation of syntax with semantics has made the generation of test cases more effective and fruitful. The next chapter will cover result of this framework along with the comparison with other existing frameworks.

# Chapter 5

# Implementation and Result Analysis

This chapter aims to experimentally evaluate the performance of SSTF by applying it on sample projects. This evaluation justifies the proposed approach as well as shows its superiority over the exisiting approaches. A prototype of SSTF was implemented in java programming language for assessing the performance of this proposed approach. Experiments were done for proving the efficiency of SSTF over the conventional[1] approaches. The execution time, number of generated tests, number of useful (runnable) test cases, as well as test coverage are the metrics for the performance measurement. SSTF was run in several java projects for conducting these metrics measurement. Alongside, same projects were used on one of the conventional syntactic approach named as Fusion [11] and the results were analyzed to get an estimation about the comparative usability of SSTF.

[1]Throughout the document, syntactic or semantic approaches are mentioned to be the 'conventional approach'

## 5.1 Environmental Setup

This section discusses the equipments used to develop the SSTF prototype and experimental procedures followed for the evaluation task. As mentioned earlier, the prototype was developed in java programming language. In order to develop the prototype, following tools were used :

- Eclipse Kepler (4.3) [49]
- javaparser version-1.0.8 [50]
- Enterprise Architect (EA) [51]
- Eclipse Plugin JUnit 4 [52]

The prototype of SSTF [53] based on Eclipse 4.3 was developed having test creation and run functionalities. Besides, configurable syntax-semantic matcher was also developed for fulfilling different users requirements of source-UML matching in test case generation. This configuration feature allows testers to decide if source and UML diagrams equivalency would effect test generation or not; and if it does, to what extent would the affect take place.
The experiments were performed on the following Desktop configuration:

- 2.20 GHz Intel Core 2 Duo Processor
- 6GB of RAM
- Windows 8 OS
- Java 7

For the assessment of the approach, four different projects [54] were selected along with the UMLs of those. Table 5.1 shows the projects with its attributes. The line of code (LOC) of source code, number of classes in the class diagram and number of sequence diagrams are the attributes. It is noticeable here that even if

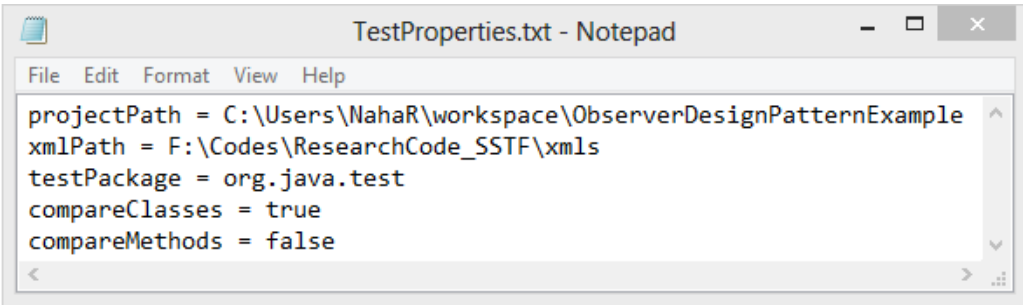



Figure 5.1: Sample Configuration File

class or sequence UMLs are not provided, the framework will still generate unit test scripts depending on the source code. In that case, syntax-semantics matching won't be done in absence of class diagram and integration tests won't be generated in absence of sequence diagrams.

Table 5.1: Experimented Projects

| Project Name | LOC | No. of Classes in Class Diagram | No. of Sequence Diagram |
|---|---|---|---|
| ObserverPatternExample | 141 | 5 | 1 |
| CalendarWithReminder | 1378 | Not Provided | 1 |
| ConnectFourGame | 1769 | Not Provided | 4 |
| Calculator | 2205 | 44 | 2 |

Some prearrangements need to be done before running SSTF on the sample project set. The source and XMLs (generated from UMLs using EA) of the projects are used as the input of the SSTF prototype. If the UMLs are not available with the source code, those can be produced by reverse engineering in Visual Paradigm Version 11.2 [55]. The projects run properties need to be included in a configuration file, where the project path, XMLs path, test package, and tester's decisions about comparing syntax-semantic classes and methods are specified. A sample configuration file is shown in Figure 5.1. A proper directory structure of XML files also needs to be maintained for program readability. The directory structure is shown in Figure 5.2.

The competency of generated test cases were measured by the execution time and ratio of runnable tests. The execution time is measured to be the efficiency and

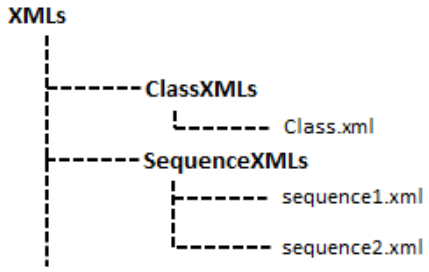



Figure 5.2: Directory Structure of XMLs

the ratio of runnable tests in the generated tests is measured to be the effectiveness of the prototype. Comparison of these metrics with Fusion [11] is done here to show the excellence of SSTF. Fusion was executed on the same projects mentioned above and then the results of SSTF and Fusion were compared.

## 5.2 Comparative Analysis

The comparative analysis of SSTF in this section, shows a significant contribution of SSTF over the existing state of the art methodologies. While the existing methodologies considered either syntax or semantics of a software, SSTF's consideration of both made a meaningful improvement in test generation. As mentioned earlier, the efficiency and effectiveness of SSTF were measured by the time required for test generation and the number of error-free compilable tests accordingly. Finally, the average satisfaction was analyzed based on the score achieved from efficiency and effectiveness.

### 5.2.1 Efficiency

To find out the execution time, SSTF was run on the selected projects as shown in Table 5.1. SSTF was run 10 times on the projects and the average time was computed. Table 5.2 shows the number of generated test cases for each project and the average execution time elapsed for this generation task. This table clearly shows that the SSTF execution time is considerably small having an average of

about 0.66 seconds to analyze and generate test cases for a project in the set. Moreover, it generates on an average 46 test cases per project in this elapsed time. For comparing the efficiency of SSTF, one of the conventional appoaches, Fusion

Table 5.2: Test Cases Generated with SSTF and Average Execution Time on the Desktop Configuration

| Project Name | No. of Generated Unit Tests | No. of Generated Integration Tests | Avg. Execution Time (sec) |
|---|---|---|---|
| ObserverPatternExample | 15 | 1 | 0.4756 |
| CalendarWithReminder | 43 | 1 | 0.5585 |
| ConnectFourGame | 30 | 4 | 0.5788 |
| Calculator | 87 | 2 | 1.0344 |

was executed on the same project set. It was again executed 10 times on the sample project set and the average time was taken as the final execution time. The difference between the execution times in Table 5.3 show that time for test generation in SSTF is fast and insignificant.

Table 5.3: Comparison of Execution Time between SSTF and Fusion

| Project Name | Generated Tests | | Avg. Time (sec) | | Time per Test | |
|---|---|---|---|---|---|---|
| | SSTF | Fusion | SSTF | Fusion | SSTF | Fusion |
| ObserverPatternExample | 16 | 10 | 0.4756 | 0.3 | 0.0297 | 0.03 |
| CalendarWithReminder | 44 | 55 | 0.5585 | 2.8 | 0.01269 | 0.0509 |
| ConnectFourGame | 34 | 35 | 0.5788 | 0.6 | 0.01702 | 0.01714 |
| Calculator | 89 | 180 | 1.0344 | 4.00 | 0.0116 | 0.022 |

Here, SSTF generates both unit and integration tests from source and UML parsing, while Fusion takes unit test scripts along with source code and generates integration tests. Still, it takes more time for Fusion to complete test generation. From the table, it can be seen that SSTF's elapsed time per tests generated is also smaller than Fusion.

Figure 5.3 and Figure 5.4 show comparison of SSTF-Fusion in execution time and SSTF execution time per test script generation accordingly. These graphs confirm that the elapsed time of SSTF is very small which make it much more efficient than other conventional approaches. Figure 5.3 shows that SSTF graph

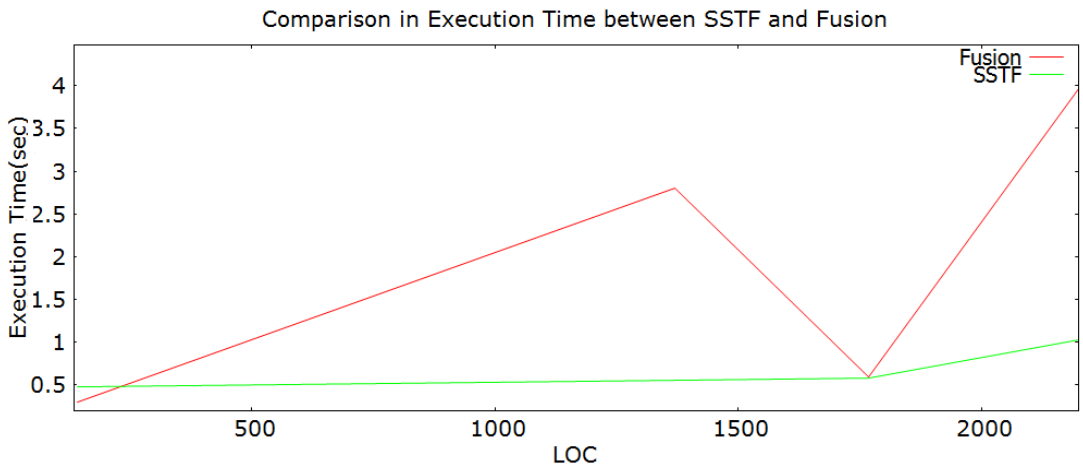



Figure 5.3: Comparison in Execution Time between SSTF and Fusion

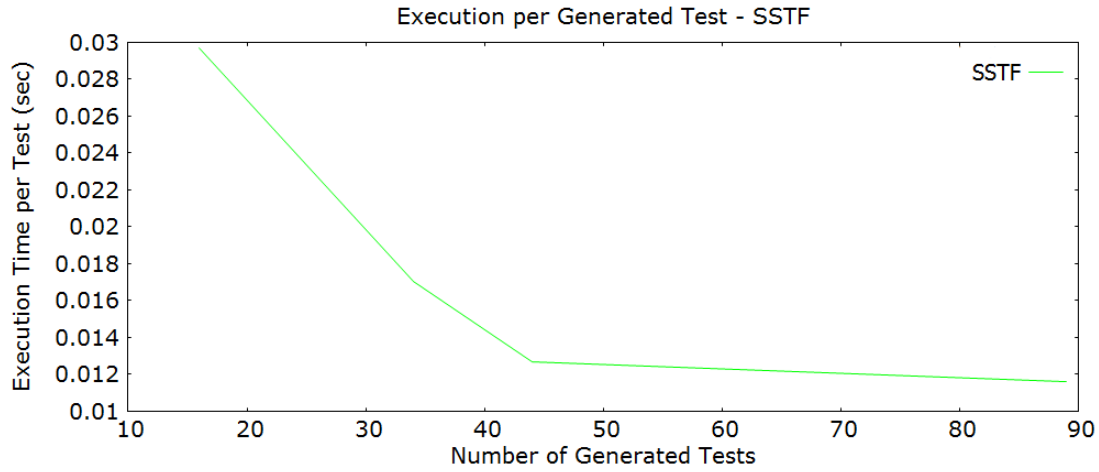



Figure 5.4: Execution Time per Generated Test - SSTF

line is pretty steady and rises slowly with LOC of source code. However, the fusion graph lines seems to be fluctuating too much as it depends on both LOC and number of inputted unit test cases. On the other hand, Figure 5.4 shows that with increase of number of generated tests, required time per test decreases.

### 5.2.2 Effectiveness

The effectiveness of SSTF is measured by the proportion of generated useful test scripts. Table 5.4 shows that, nearly 100% of generated test cases can be compiled and run instantly for the proposed framework. Only for project 'Calculator' the

ratio is 98.88% as one of the test cases could not be compiled. The reason behind this is that the framework tried to instantiate an Interface[2] which caused a compilation error. This problem can be solved by identifying the implemented class of that interface and instantiating that class. This information is also available in UML class diagram and so can be fixed. However, the other cases were successfully handled providing 100% of runnability ratio for other projects. This assures the effectiveness of SSTF in test generation.

Table 5.4: Test Cases Generated with SSTF and Ratio of Runnable Tests

| Project Name | No. of Generated Test Cases | No. of Compilable and Runnable Tests | Ratio (%) |
|---|---|---|---|
| ObserverPatternExample | 16 | 16 | 100% |
| CalendarWithReminder | 44 | 44 | 100% |
| ConnectFourGame | 31 | 31 | 100% |
| Calculator | 89 | 88 | 98.88% |

On the other hand, on an average of 60% of the generated test cases of Fusion are runnable [11], while the remaining 40% needs manual modification to become executable. However, for the selected project set the success rate was 73.89% on an average. Table 5.5 shows the ratio of the runnable test scripts of the set. It is noticeable here that there is an extra column named 'No. of Skipped Tests'. It is the number of provided unit tests for which no test cases was created by Fusion. This is unacceptable and is considered as test generation failure. Thus, the measurement of success rate has been done by the following equation -

$$SuccessRate = \frac{T_{compilable}}{T_{generated} + T_{skipped}}$$

Figure 5.5 shows the comparison between SSTF and Fusion on the basis of usefulness of generated test scripts. The figure shows that the SSTF line is always near to 100% while Fusion line is rising. It is noticeable that, SSTF produces 99.72% runnable test scripts on average for this project set.

[2]Interfaces cannot be instantiated

Table 5.5: Test Cases Generated with Fusion and Ratio of Runnable Tests

| Project Name | No. of Generated Test Cases | No. of Compilable Test Cases | No. of Skipped Tests | Success Rate (%) |
|---|---|---|---|---|
| ObserverPatternExample | 10 | 5 | - | 50% |
| CalendarWithReminder | 55 | 55 | 13 | 80.88% |
| ConnectFourGame | 35 | 35 | 10 | 77.78% |
| Calculator | 180 | 166 | 11 | 86.91% |

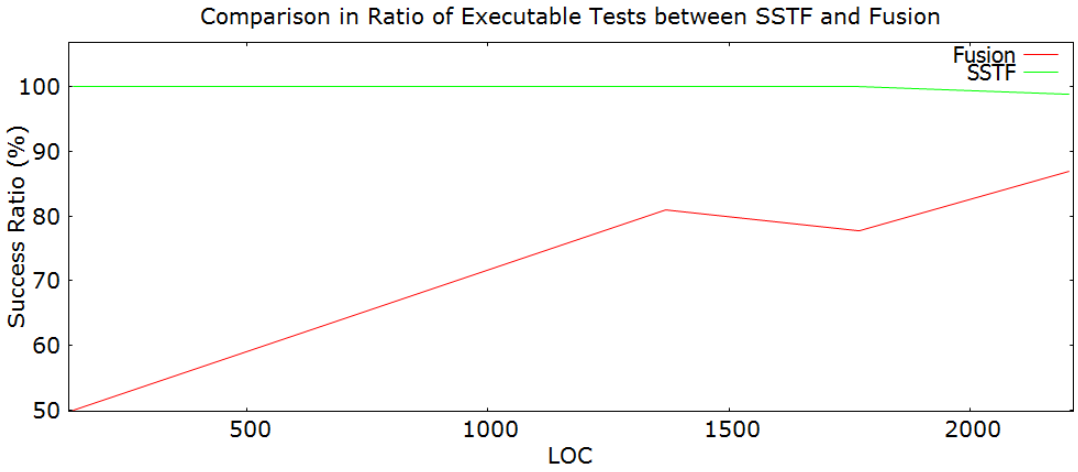



Figure 5.5: Comparison in Ratio of Executable Tests between SSTF and Fusion

### 5.2.3 Satisfaction

The satisfaction of testers in using a test automation tool depends on a number of metrics. Execution time, runnability of test scripts and test coverage are the metrics to define fruitfulness of tests. It is already showed that SSTF's performance is impressive with respect to execution time and runnability of test scripts. About the test coverage, it also can be achieved by this framework. In SSTF, for every public[3] methods in all classes, one unit test is generated and for every provided sequence diagram, one integration test is generated. This shows the fact that full test coverage can be achieved by these generated test cases if proper test data is synchronized with it. Thus, achieving test coverage seems to be more appropriate in test data generation than test script generation. However, the measurement

[3]Only public methods can be tested by a test case, as private methods cannot even be accessed by other classes except the parent class

of test coverage is not given priority in this research as test coverage is loosely correlated with test suite effectiveness [56].

## 5.3 Discussion of Result

This section justifies the reasons behind the obtained results from SSTF. Unlike other conventional approaches, the proposed approach aims to generate both unit and integration test cases considering software syntax and semantics at the same time. This is why the generated test scripts does not suffer from the limitations of the solely considered syntactic or semantic approach. However, in one case SSTF seemed to generate non-compilable test which is when it tried to instantiate Interface. This is not a limitation of the approach, but the framework. This limitation can be omitted by considering the interface to class implementation information provided by class diagram.

The reason behind the generation of faulty test cases in semantic approach is that, it does not have proper syntax information for creating the test scripts. As a result, the generated scripts from UMLs does not synchronize with source code which makes the scripts non-compilable. Similarly, although syntactic approach contains script generation syntax, it fails to generate proper integration tests because of the lack of software semantic knowledge.

Fusion generates integration test cases while software source code and unit test scripts are available. 60% of the Fusion generated test scripts are runnable leaving the remaining 40% unusable [11]. The reason behind generation of these 40% unsuccessful test scripts is that, syntax parsing is a complex task making it difficult to find out software integration information from inside code. This task has been made way easier by the incorporation of sequence diagram with software syntax in this research. In our research Fusion seems to work more effectively as the projects are relatively small and created in controlled environment.

The less erroneous generated test scripts proof the competency of the approach proposed. While unit test generation was done by the syntax analysis, integration test generation was simplified by the incorporation of sequence diagram with it. Moreover, class diagram was integrated for the syntax-semantics comparison purpose, so that the syntactic and semantic knowledge of software can be synchronized together before the test generation.

## 5.4 Summary

This chapter intends to demonstrate the implementation and result analysis of SSTF. A prototype of SSTF is developed using java. The prototype results are analyzed in comparison with another conventional approach, Fusion. The analysis shows that SSTF is more useful than the state of the art test generation frameworks. 99.72% of generated tests of SSTF are executable. Elapsed time for test generation is also very small. To be specific, SSTF is more time efficient, test script effective and satisfactory than the existing ones. The next chapter shows a case study on one of the sample projects.

# Chapter 6

# A Case Study on Project 'ObserverPatternExample'

The result analysis of the developed prototype of SSTF shows the proficiency of the framework. However, a step-by-step study may increase the understanding of the framework as well as justify the reason behind the improved results than conventional approaches. Thereby, this chapter provides a phase-to-phase analysis of test generation input processing by different components of the framework. It also shows the processed outcome of different phases in forms of snapshots.

## 6.1 Phases of the Case Study

For an assessment of the competency of the approach, the SSTF was used on a simple java project illustrating observer pattern. The observer pattern is a simple design pattern in which an object, named as subject, maintains a list of its dependents, named observers, and notifies those whenever any state change occurs, generally by calling one of their methods [57]. The example project has a class called Person as observer class, a class called Product as subject class, and the Observer and Subject Interfaces.

The main phases of the case study are: (1) Analysis of Source Code, (2) Getting XMLs from UMLs, (3) Analysis of XMLs, (4) Matching Syntax with Semantics, (5) Generation of Unit Test, and (6) Generation of Integration Test.

### 6.1.1 Analysis of Source Code

Figure 6.1 and Figure 6.2 show snapshots of the source code of the Person and

```
public class Person implements Observer {

    String personName;

    public Person(String personName) {
        this.personName = personName;
    }

    public String getPersonName() {
        return personName;
    }

    public void setPersonName(String personName) {
        this.personName = personName;
    }

    public void update(String availabiliy) {

        System.out.println(personName +
                ", Product is now " + availabiliy);
    }
}
```

Figure 6.1: Source Code Example (Class: Person)

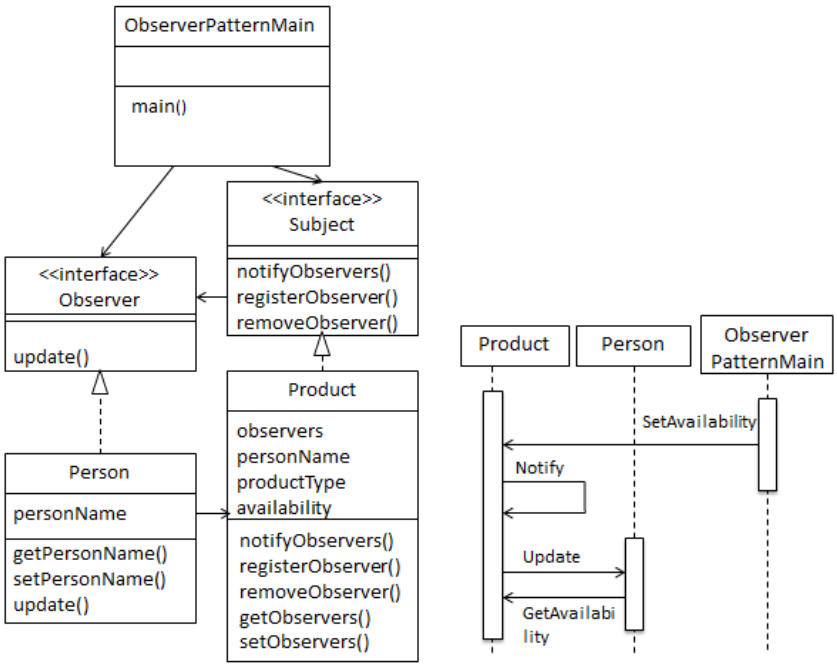
```
public class Product implements Subject {

    private ArrayList<Observer> observers
              = new ArrayList<Observer>();
    private String productName;
    private String productType;
    String availability;

    public Product(String productName,
        String productType, String availability) {
        super();
        this.productName = productName;
        this.productType = productType;
        this.availability = availability;
    }
    public void notifyObservers() {
        for (Observer ob : observers) {
            ob.update(this.availability);
        }
    }
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
```

Figure 6.2: Source Code Example (Class: Product)

Product class accordingly. The Source Reader takes this project source location as input and reads the java files to be processed later by Syntax Parser in second layer. The Syntax Parser parses the source for gathering the syntactic knowledge. It identifies significant information from inside source. The class name, constructors, package declaration, class imports are stored along with the method details such as method name, body, parameters, annotation and exception thrown.

### 6.1.2 Getting XMLs from UMLs

The project UML diagrams are inputted in UML Reader. These are then converted to XML using the XML Converter Component. Enterprise Architect is used for




(a) Class diagram (b) Sequence diagram

Figure 6.3: The UML Diagrams of the Sample Project

this conversion purpose [51]. The class and sequence diagrams are shown in Figure 6.3 and a portion of the produced XMLs are illustrated in Figure 6.4.

### 6.1.3 Analysis of XMLs

The second layer takes the source and the produced XML as input. The XML Reader of the layer takes the XMLs and sends those to appropriate analyzer. The Class Information Extractor which is the class diagram analyzer, analyzes XML of the class diagram and identifies the class methods, variables as well as class association. The <owedAttribute> tags of the class XML refer class variables and the <owedOperation> tags refer class methods. The Inter-Class Method Call Sequence is the sequence diagram analyzer and extracts sequence diagram information similarly from their XML by parsing appropriate tags in it.

```
<packagedElement xmi:type="uml:Class" xmi:id="EAID_25DB71D2_C4DF_412a_8488_0038F9C1FB3A" name="Person" visibility="public">
    <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_8A76C062_7F0F_4eed_B684_D6A3338BAE92" name="personName" visibility="private"
        isStatic="false" isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="EAID_LI000001_7F0F_4eed_B684_D6A3338BAE92" value="1"/>
        <upperValue xmi:type="uml:LiteralInteger" xmi:id="EAID_LI000002_7F0F_4eed_B684_D6A3338BAE92" value="1"/>
        <type xmi:idref="EAJava_String"/>
    </ownedAttribute>
    <ownedOperation xmi:id="EAID_004CF887_09D2_4cdc_A631_8607E65C0CE6" name="getPersonName" visibility="public" concurrency="sequential"/>
    <ownedOperation xmi:id="EAID_787223EC_1888_443c_ACD3_E36EE5274CB2" name="setPersonName" visibility="public" concurrency="sequential">
        <ownedParameter xmi:id="EAID_RT000000_1888_443c_ACD3_E36EE5274CB2" name="return" direction="return" type="EAJava_void"/>
    </ownedOperation>
    <ownedOperation xmi:id="EAID_2EBA5E9C_A27F_445e_AFDF_882B49AA55FF" name="update" visibility="public" concurrency="sequential">
        <ownedParameter xmi:id="EAID_RT000000_A27F_445e_AFDF_882B49AA55FF" name="return" direction="return" type="EAJava_void"/>
    </ownedOperation>
    <generalization xmi:type="uml:Generalization" xmi:id="EAID_BB73DD25_439E_4b15_AE39_E1F3A08814F7" general="EAID_03BF37E7_AED7_4265_B746
</packagedElement>
```

(a) Class XML (partial: Person class)

```
<connector xmi:idref="EAID_BD63EEAD_C204_4de6_B589_76D8BBDE3170">
    <source xmi:idref="EAID_EB85FD4D_6106_480b_BFCE_AC675D9483D9">
        <model ea_localid="3" type="Sequence" name="Person"/>
        <role visibility="Public" targetScope="instance"/>
        <type containment="Unspecified"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
        <style value="Union=0;Derived=0;AllowDuplicates=0;Owned=0;Navigable=Non-Navigable;"/>
        <documentation/>
        <xrefs/>
        <tags/>
    </source>
    <target xmi:idref="EAID_AACF65BB_FA20_481f_8EA4_685088E05401">
        <model ea_localid="2" type="Sequence" name="Product"/>
        <role visibility="Public" targetScope="instance"/>
        <type aggregation="none" containment="Unspecified"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
        <style value="Union=0;Derived=0;AllowDuplicates=0;Owned=0;Navigable=Navigable;"/>
        <documentation/>
        <xrefs/>
        <tags/>
    </target>
    <model ea_localid="6"/>
    <properties name="GetAvailability" ea_type="Sequence" direction="Source -&gt; Destination"/>
    <documentation/>
    <appearance linemode="1" linecolor="-1" linewidth="0" seqno="4" headStyle="0" lineStyle="0"/>
    <labels mt="GetAvailability()"/>
    <extendedProperties stateflags="Activation=0;" virtualInheritance="0" diagram="EAID_D835BB34_
    <style/>
    <xrefs/>
    <tags/>
</connector>
```

(b) Sequence XML (partial: GetAvailaility call)

Figure 6.4: Portions of Generated XMLs

### 6.1.4 Matching Syntax with Semantics

These gathered information are then compared in Syntax Semantic Matcher to check if there is any inconsistency between the syntax and semantic information. For assessing the efficiency of this component, some extra classes and methods were added in source. And some more were added in class diagram. These mismatched

classes and methods were successfully identified by the component. A text file was generated reporting the inconsistency in the syntax-semantic information.

### 6.1.5 Generation of Unit Test

The semantic class information along with the class syntax are inputted in the Unit Test Generator. Test cases are generated for the matched methods of the matched classes only. The details of the methods are already present in the syntax information. These methods are called with appropriate parameter set and thus, a test method stub is generated for each method call. The class instantiation is done in the setUp stub from class constructor information inside syntax.

```
@Before
public void setUp() {
    product = new Product("test","test","test");
}

@Test
public void registerObserverTest() {
    Observer mockVar0;
    mockVar0 = EasyMock.createMock(Observer.class)
    product.registerObserver(mockVar0);
}

@Test
public void removeObserverTest() {
    Observer mockVar0;
    mockVar0 = EasyMock.createMock(Observer.class)
    product.removeObserver(mockVar0);
}

@Test
public void setProductNameTest() {
    product.setProductName("test");
}
```

Figure 6.5: Unit Test Example

```
private ObserverPatternMain observerPatternMain;
private Person person;
private Product product;

@Before
public void setUp() {
    observerPatternMain = new ObserverPatternMain();
    person = new Person("test");
    product = new Product("test","test","test");
}

@Test
public void NotificationTest() {
    product.setAvailability("test");
    product.notifyObservers();
    person.update("test");
    product.getAvailability();
}
```

Figure 6.6: Integration Test Example

### 6.1.6 Generation of Integration Test

For the generation of integration test cases, sequence diagrams contain the most significant information. The method call sequence got from these are gathered together for the generation. Again the syntax of method calls are acquired from the syntactic information. These syntax of method calls are used in places of each method call of the sequence. One sequence diagram generates one integration test.

A sample unit and a sample integration test suite snapshots are shown in Figure 6.5 and Figure 6.6.

## 6.2 Summary

This case study shows SSTF competence in generating unit and integration tests. The stepwise demonstration of the approach makes it clear why SSTF generated test execution ratio is high. The matching of syntax and semantics also assures that the generated tests do not suffer from the effects of backdated UMLs or code segment. On the other hand, the use of sequence diagram for generation of integration test reduces efforts for extracting method call sequence from source code. The next chapter concludes this thesis after providing a proper future direction.

# Chapter 7

# Conclusion

Unlike conventional test generation approaches, this research proposes to incorporate software syntax and semantics. A framework named SSTF is proposed for this purpose. SSTF uses UMLs for obtaining software internal structure and interactions between its classes, and source code for deriving test scripts' body. In this chapter, the document is concluded by a profound but brief discussion of the research along with the future direction.

## 7.1 Discussion

This thesis introduces a testing framework named SSTF for the generation of unit and integration test cases automatically using software semantics and syntax. Most of the automated test generation techniques in the literature consider either syntactic or semantic approach. Those individual approaches do not always produce effective test cases as the semantic one lacks syntactic knowledge which is required for the test syntax creation, and the syntactic one misses semantic information, which is needed for understanding the software. Thus, the incorporation of semantics and syntax of software in test construction can improve the quality of generated test scripts. This is the core idea of this work.

The User End, Service and Test Run layers of the framework operate together for this generation task. While User End Layer processes user inputs, Service Layer does the identification of semantics and syntax and produce test scripts incorporating those. This incorporation reduces the probability of generating ineffective and in-executable test cases. Finally, the Test Run Center compile and run the generated scripts.

The experiments conducted on sample projects show that the elapsed time of SSTF is 0.66 seconds on average and 99.72% scripts are runnable which is an immense improvement compared to another test generation tool, Fusion. A case study on a sample java project is also shown here to assess the competence of the approach and it has accomplished to construct proper test scripts.

## 7.2 Future Work

The idea of incorporating software syntax with semantics opens a number of directions for future research. It can be used not only for test script generation, but also for test data generation, test case prioritization etc. The integration of proper test data with the generated test scripts of SSTF can also be a future issue. This integration can lead to the achievement of test coverage.

The current SSTF implementation evaluated the performance of the approach on java projects. The future challenge lies in generation of test scripts for different platforms rather than java only. Moreover, the Syntax-Semantics Matcher component of SSTF also provides some future directions. This matcher identifies the syntax-semantics inconsistency by checking the names of the classes or methods. Thus, there is a scope of improvement which can be done by introducing a proper inconsistency checker.

# Bibliography


[1] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *Proc. of the 40th Annual Hawaii International Conference on System Sciences (HICSS)*, p. 261a, Jan. 2007.

[2] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Reinhold, 2nd ed ed., 1990.

[3] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional ©2009, 2009.

[4] K. Conroy, M. Grechanik, M. Hellige, E. Liongosari, and Q. Xie, "Automatic test generation from gui applications for testing web services," in *Proc. of the 2007 IEEE Software Maintenance (ICSM)*, pp. 345–354, Oct. 2007.

[5] E. G. Cartaxo, F. Neto, and P. Machado, "Test case generation by means of uml sequence diagrams and labeled transition systems," in *Proc. of the 2007 IEEE Systems, Man and Cybernetics (ISIC)*, pp. 1292–1297, Oct. 2007.

[6] A. Javed, P. Strooper, and G. Watson, "Automated generation of test cases using model-driven architecture," in *Proc. of the 2nd IEEE International Workshop on Automation of Software Test, (AST '07)*, p. 3, May 2007.

[7] E. Enoiu, D. Sundmark, and P. Pettersson, "Model-based test suite generation for function block diagrams using the uppaal model checker," in *Proc. of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 158–167, Mar. 2013.

[8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 213–223, June 2005.

[9] E. Duclos, S. Digabel, Y. Gueheneuc, and B. Adams, "Acre: An automated aspect creator for testing c++ applications," in *Proc. of the 17th IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 121–130, Mar. 2013.

[10] G. Fix, "The design of an automated unit test code generation system," in *Proc. of the 6th IEEE International Conference on Information Technology: New Generations (ITNG'09)*, pp. 743–747, Apr. 2009.

[11] M. Pezz, K. Rubinov, and J. Wuttke, "Generating effective integration test cases from unit ones," in *Proc. of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 11–20, Mar. 2013.

[12] E. Diaz, J. Tuya, and R. Blanco, "A modular tool for automated coverage in software testing," in *Proc. of the 11th IEEE Annual International Workshop on Software Technology and Engineering Practice*, pp. 241–246, Sept. 2003.

[13] M. Chen, X. Qiu, and X. Li, "Automatic test case generation for uml activity diagrams," in *Proc. of the 2006 International Workshop on Automation of Software Test (AST'06)*, (Shanghai, China), pp. 2–8, May 2006.

[14] C. Nebut, F. Fleurey, Y. Traon, and J. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[15] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[16] P. C. Company, *Dr. Dobb's Journal of Software Tools for the Professional Programmer: 1988: Jan-Jun*, vol. 13. M & T Pub., 1988.

[17] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[18] D. Gelperin and B. Hetzel, "The growth of software testing," *Communications of the ACM*, vol. 31, no. 6, pp. 687–695, 1988.

[19] A. Turing, "Checking a large routine," in *The early British computer conferences*, pp. 70–72, MIT Press, 1989.

[20] C. Baker, "Review of d.d. mccracken's "digital computer programming"." Mathematical Tables and Other Aids to Computation 11, 60, Oct. 1957.

[21] M. S. Deutsch, *Software verification and validation: Realistic project approaches*. Prentice Hall PTR, 1981.

[22] W. E. Howden, "Life-cycle software validation," *Computer*, vol. 15, no. 2, pp. 71–78, 1982.

[23] "Guideline for lifecycle validation, verification and testing of computer software." Federal Information Processing Standards, FIPS PUB101, National Bureau of Standards, 1983.

[24] R. Osherove, *The art of unit testing*. mitp, 2010.

[25] K. Beck, "Simple smalltalk testing: With patterns," *The Smalltalk Report*, vol. 4, no. 2, pp. 16–18, 1994.

[26] P. Runeson, "A survey of unit testing practices," *Software, IEEE*, vol. 23, no. 4, pp. 22–29, 2006.

[27] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation0," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[28] F. P. Brooks, *The mythical man-month*, vol. 1995. Addison-Wesley Reading, MA, 1975.

[29] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the Pieces Together, Addison-Westley*, 2001.

[30] H.-G. Gross, *Component-based software testing with UML*, vol. 44. Springer, 2005.

[31] L. Briand and Y. Labiche, "A uml-based approach to system testing," *Software and Systems Modeling*, vol. 1, no. 1, pp. 10–42, 2002.

[32] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

[33] J. J. Marciniak and A. Shumskas, *Acceptance Testing*. Wiley Online Library, 1994.

[34] "Survey. 1996." Centerline Software , Inc. CenterLine is a software testing tool and automation company in Cambridge, Massachusetts.

[35] L. Hayes, "Evolution of automated software testing," *Automated Software Testing*, pp. 14–20, 2009.

[36] A. V. Lamsweerde, "Building formal requirements models for reliable software." Springer Berlin Heidelberg, 2001.

[37] R. Heckel and M. Lohmann, "Towards model-driven testing." TACoS'03, International Workshop on Test and Analysis of Component-Based Systems, Apr. 2003.

[38] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecturepractice and promise*. Addison-Wesley, 2003.

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

[40] D. R. Wallace and R. U. Fujii, "Software verification and validation: An overview," *IEEE Computer Society*, vol. 6, no. 3, pp. 10–17, 1989.

[41] D. DIO, "Function block diagram," *Citeseer*.

[42] "Uppaal." `http://www.uppaal.org/`. Online; accessed 27 Sept. 2014.

[43] F. Glover, *Tabu Search*. Kluwer Academic Publishers Norwell, MA, USA ©1997, 1997.

[44] R. Marta, M. Lagunab, and F. Gloverb, "Principles of scatter search," *European Journal of Operational Research*, vol. 169, pp. 359–3725, March 2006.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[46] A. V. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN NOTICES*, vol. 35, pp. 26–36, 2000.

[47] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *Proc. of the 23rd IEEE International Conference on Automated Software Engineering, ASE'08*, pp. 407–410, Sept. 2008.

[48] M. Sharma and B. Chandra, "Automatic generation of test suites from decision table - theory and implementation," in *Proc. of the 5th IEEE International Conference on Software Engineering Advances (ICSEA)*, pp. 459–464, Aug. 2010.

[49] "Eclipse.org - kepler simultaneous release : Highlights." `http://eclipse.org/kepler/`. Online; accessed 3 Oct. 2013.

[50] "javaparser." `https://code.google.com/p/javaparser/`. Online; accessed 9 July 2014.

[51] "Enterprise architect - uml design tools and uml case tools for software development." `http://www.sparxsystems.com/products/ea/`. Online; accessed 1 Sept. 2013.

[52] "Junit - about." `http://junit.org/`. Online; accessed 29 Nov. 2014.

[53] "Nadiait/sstf-github." `https://github.com/NadiaIT/SSTF`. Online; accessed 2 Nov. 2014.

[54] "Sampletestprojects-java-github." `https://github.com/NadiaIT/SampleTestProjects-Java`. Online; accessed 2 Nov. 2014.

[55] "Software design tools for agile teams, with uml, bpmn and more." `http://www.visual-paradigm.com/`. Online; accessed 29 July. 2014.

[56] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. of the 36th International Conference on Software Engineering (ICSE)*, (Hyderabad, India), pp. 435–445, May 2014.

[57] "Observer pattern — object oriented design." `http://www.oodesign.com/observer-pattern.html`. Online; accessed 29 July 2014.