# Software semantics and syntax as a tool for automated test generation

Nadia Nahar* and Kazi Sakib

Institute of Information Technology,
University of Dhaka,
Dhaka, Bangladesh
Email: nadia@iit.du.ac.bd
Email: sakib@iit.du.ac.bd
*Corresponding author

**Abstract:** Test automation saves time and cost by digitising test generation and execution. The existing techniques fail to produce effective and compliable test scripts for not considering both syntactic and semantic information. The proposed three-layer architecture incorporates these information for generation of proper test scripts. It analyses the source code for extracting syntax and UML diagrams for obtaining semantics. Class methods and sequence of calls are extracted from UMLs, and syntax for class instantiations and method calls are accumulated from source code to generate unit and integration tests. A case study as well as experiments, conducted on sample Java projects, conform the competence of the generation process along with the generated test scripts.

**Keywords:** software testing; test automation; automatic test generation; unit testing; integration testing.

**Reference** to this paper should be made as follows: Nahar, N. and Sakib, K. (2017) 'Software semantics and syntax as a tool for automated test generation', *Int. J. Critical Computer-Based Systems*, Vol. 7, No. 4, pp.369–396.

**Biographical notes:** Nadia Nahar is a Lecturer at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. She pursued her Master of Science in Software Engineering (MSSE) and Bachelor of Science in Software Engineering (BSSE) from the same institution. She was a Gold Medalist for attaining top score in her class. As a student, her efforts have earned awards from different national and international software and programming competitions, project show casings as well as publications in various international conferences. She has the experiences of working both in industry and academia. Her core areas of interest are software engineering, web technologies, systems and security.

Kazi Sakib is a Professor at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. He received his PhD in Computer Science at the School of Computer Science and Information Technology, RMIT University. His research interests include software engineering, cloud computing, software testing, software maintenance, etc. He is an author of a great deal of research studies published at national and international journals as well as conference proceedings.

This paper is a revised and expanded version of a paper entitled 'SSTF: a novel automated test generation framework using software semantics and syntax' presented at the 17th International Conference on Computer and Information Technology (ICCIT), Bangladesh, 22–23 December 2014.

Copyright © 2017 Inderscience Enterprises Ltd.

## 1 Introduction

Automated testing is a process where software is tested without any manual input, analysis or evaluation (Leitner et al., 2007). In this type of testing, software functionality is tested as a verification testing. 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 (Beizer, 1990). Manual testing is a lengthy process as testers use test plans, cases or scenarios to test software by using their profound knowledge on software semantics and syntax. Automated testing saves time and cost by automating this whole process of test generation and execution (Dustin et al., 2009). However, understanding the program semantically and incorporating syntax with it without human interaction may produce undesirable results such as redundant or non-executable test cases (Nahar and Sakib, 2014).

### *1.1 Motivation*

Generally two approaches are used for test suite automation – software requirements specification (SRS) analysis and source code parsing. In SRS analysis, test cases are automatically generated from requirements, various UML diagrams – class, state, sequence diagrams and also from GUI screens (Nebut et al., 2006; Chen et al., 2006; Conroy et al., 2007; Cartaxo et al., 2007; Javed et al., 2007; Utting and Legeard, 2007; Enoiu et al., 2013; Sharma and Biswas, 2014; Singi et al., 2015; Khatun and Sakib, 2016). This approach can also be referred as semantic approach because the software specification is considered here. Another approach of automated testing is the source code parsing which can also be referred as syntactic approach as the software syntactical information (i.e., the construction detail in source code) is used here for test generation (Diaz et al., 2003; Godefroid et al., 2005; Pacheco et al., 2007; Fix, 2009; Fraser and Arcuri, 2011; Duclos et al., 2013; Pezze et al., 2013; Arcuri et al., 2014). In this approach, code parsing is done to identify the class relations and the method call sequences. This extracted information is used to form the syntax of the test cases and generate those according to the control flow of the source code.

The semantic approach does not always produce effective or runnable 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. Nebut et al. (2006) proposed a semantic test generation approach using use cases and sequence diagrams and Singi et al. (2015) used visual requirement specifications or prototypes as semantics for automated test generation. Both the papers suffered from the inherent drawbacks of semantic approach. On the other hand, parsing done in syntactic approach is not enough to identify the semantic information hidden inside the code. Now as the semantic information is missing, test cases are generated using randomisation or other approaches without considering the software semantics. It often results in generation of redundant or non-executable test suites. Pacheco et al. (2007) proposed a test generation technique named Randoop which is an example of redundancy in the automatically generated test scripts by syntactic approach. Another syntactic automated test case generation framework called Fusion was proposed by Pezze

et al. (2013) which resulted in generation of 40% non-executable test cases due to the mentioned drawbacks.

### *1.2 Research questions*

The existing researches on automated test case generation focus either on semantic or on syntactic approach. The shortcomings of existing approaches have been discussed above. An integration of these approaches is needed to overcome the drawbacks. Thus, this leads to the 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 where semantic information will be extracted from UML models, and software syntax will 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 code and UML diagrams of software as input and produce test cases of the software as output.

### *1.3 Contribution and achievement*

To answer the research question, a test generation framework is proposed which uses the information extracted from UML diagrams and source code. Unlike the state-of-the-art approaches, this framework incorporates syntax and semantic information together with an intention to increase the executability rate of the generated test scripts. It works in three layers based on the three categories of tasks such as input processing, test generation and test execution. The layers are – input layer, service layer and test run layer. The input layer consists of UML reader (UR), XML converter (XC) and source reader (SR) 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 Input Layer and constructs 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 initialisation and method call syntax retrieved from source. Finally, the test run layer provides the test editor, compiler and runner.

A case study has been conducted here for the assessment of the proposed approach. The case study is carried out on a Java project. 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 Input 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 run by the test run layer. This case study evaluates the approach, that the incorporation of semantics and syntax can lead to the generation of executable test scripts.

The evaluation of the proposed framework has been done by applying the framework on four sample projects and analyzing the results. These projects of the sample set are

written in Java. For the comparative analysis, an existing state-of-the-art semantic approach of Sharma and Biswas (2014) and syntactice approach, fusion (Pezze et al., 2013) are also applied on the same sample set. The results of all 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.

## 2 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 parsing – these two approaches can be termed as semantic and syntactic approach accordingly. In SRS analysis, which deals with the software semantics, requirements, various UMLs – class, state, sequence diagrams as well as GUI screens are analyzed (Nebut et al., 2006; Chen et al., 2006; Conroy et al., 2007; Cartaxo et al., 2007; Javed et al., 2007; Utting and Legeard, 2007; Enoiu et al., 2013; Sharma and Biswas, 2014; Singi et al., 2015). Similarly, source code parsing or the so called syntactic approach uses the software syntax for the test generation (Diaz et al., 2003; Godefroid et al., 2005; Pacheco et al., 2007; Fix, 2009; Fraser and Arcuri, 2011; Duclos et al., 2013; Pezze et al., 2013; Arcuri et al., 2014). Some authors have also focused on regression test generation (Taneja and Xie, 2008) and some have brought other concepts like user interaction, decision table (Sharma and Chandra, 2010) for the generation of unit test cases. There are also other testing related works, for example, web testing (Rudolf et al., 2002), that are out of scope of this paper.

### *2.1 Semantic approach*

The literature of automated test generation by semantic approach is a rich field. Nebut et al. (2006) proposed an approach for the automation in the context of object oriented (OO) software. A complete and automated chain for test cases was derived from formalised requirements (Lamsweerde, 2001) of embedded OO software here. The approach was based on use case models unraveling the ambiguities of the requirements written in natural language (NL). 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 performed next. Automation of the test scenario generation was done by replacing each use case with a sequence diagram. However, the source code of applications may not be completely matched with the use cases or sequence diagrams; so the generated test cases from those can be non-executable.

Javed et al. (2007) established a method that uses the model transformation technology of model driven architecture (MDA) to generate unit test cases by using sequence diagrams. The generation task was done in two levels. Firstly, test cases were generated from sequence of method calls in sequence diagrams and were selected by the tester. Then the test results were 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. However, UML diagrams are designed in the early stage of software development process so as the sequence diagrams. Now, as the agile development process is taking over other software development models, the requirement and feature changes have become a common phenomenon. In this situation, The generated test cases from the UMLs may become outdated for testing.

Another method for model-based test generation is described by Enoiu et al. (2013). It was for the safety-critical embedded applications implemented in a programming language such as function block diagram (FBD). The approach was based on functional and timing behaviour models and a model checker was used to automate generation of test suites. Firstly, transformation of FBD programs into timed automata models was done. Then UPPAAL (2014) 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. However, in addition, test generation to support integration testing could also be considered.

Singi et al. (2015) proposed a model-based approach for automated test case generation from visual requirement specifications (also known as prototypes). At first, models were created from the prototypes based on a proposed graph based modelling technique. Then, for the test case generation, the test paths were identified, and the semantics of the ‘nodes’ [behavioural user interface components (BUIC)] and ‘edges’ (behavioural events) of the test paths were analyzed. Finally, The test paths were transformed into test cases using the semantics of BUICs and events. However, the semantics cannot be fully relied on to produce test scripts as the script syntax depends on the software source code.

An approach to automatically generate test cases during the early phases of software development was proposed by Sharma and Biswas (2014). The tests were generated from a logical form of requirements specifications, that is, the courteous logic representation. Here, the courteous logic representations were generated semi-automatically using a modified semantic head-driven approach for NL Generation. For this, first the pivot elements for the input rule was identified as rule-heads. Next the body of the rule was considered, which can simply be a predicate or clause (like rule-head) or conjunction of two or more predicates. Test cases were laid out for null checks, invalid and valid values of the variables. However, as the tests were generated in the early phase of development using the specification only, the tests might not conform with the code. Also, the test generation approach covered only null and invalid values which skipped other different effective test cases.

## *2.2 Syntactic approach*

A prominent literature of the syntactic approach proposed a tool for automatic generation of test cases from source code named as Randoop (Pacheco et al., 2007). Randoop corresponds to the random generation of unit test cases. The process incorporates feedback-directed approach for improving the randomised generation. Firstly, the class instantiation and method call sequences were generated randomly. Then, valid sequences were selected after executing and filtering (e.g., equality of objects, null references, exceptions, etc.) those. The tool tended to generate simple test cases with low readability making it difficult for the testers to edit or change. Moreover, it was ineffective in generation of valid method call sequences while a large number of sequences were

possible due to its random nature, and redundancy in the generated test scripts unnecessarily increased the number or size of generated scripts.

Fix (2009) presented the design and implementation of a framework for semi-automated unit test code generation. The framework started with a program that extracted method information (method name, input-output parameters, and return type) from the system under test (SUT). An XML metadata file was produced according to the information, that was used to create an XSLT transformation style sheet file manually. This XSLT file contained the logic that generated the unit tests for the methods of the SUT. The framework proposed a simple generator program which accepted the XML metadata file and the XSLT transformation instructions as inputs, to produce the unit tests. However, for full automation of the semi-automated process, XSLT files need to be generated automatically, which can be supported by provision of software semantic information.

A new approach to use aspect-oriented programming (AOP) (Kiczales et al., 1997) for testing was proposed by Duclos et al. (2013). An automated aspect creator named ACRE was presented here to perform memory, invariant and interferences testing for software programs written in C++. ACRE used a domain-specific language (DSL) (Deursen et al., 2000), which were 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 did not modify the programâŁ™s behaviour. ACRE parsed the DSL statements and automatically generated appropriate aspects that were then weaved into the source code to identify bugs due to memory leaks, incorrect algorithm implementation, or interference among threads. However, the found bugs of ACRE may be caused by the aspects added to the source code, which means the tool to detect bug in a program can itself create bugs in it. This can be avoided by incorporation of additional software semantic information (memory size, algorithm information, etc.), dealing with those bugs.

An approach to generate complex or integration test cases using simple or unit ones has been proposed by Pezze et al. (2013) named as fusion. The key observation of the paper was that unit and integration, both test cases were produced from method calls which work as the atom of those. Unit tests contained 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 was the idea of combining initialisation and execution sequences of unit test cases to generate integration test cases. However, the approach used for the test generation is syntactic approach and fails to completely extract requirements information from inside code, producing only about 60% compliable and executable test cases.

### *2.3 Other approaches*

Regression test generation has also been emphasised by some authors. Taneja and Xie (2008) presented an automated regression unit test generator called DiffGen for Java programs. Differences between two versions of a given Java class were evaluated by checking observable outputs and receiver object states first. The detected behavioural differences 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 and Chandra (2010) to test suite auto-generation. A generic novel framework was proposed here which processed decision table for test generation, and elimination of redundancy in generated tests. 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.

These research address the importance of automated test generation. Although various automation frameworks have been proposed throughout the years, there is still room for improvements. The syntactic approaches of test generation results in redundancy and non-compilability of test scripts because of difficulties in understanding software behaviour. The semantic approaches lack syntax for producing test scripts and again lead to generation of non-compliable scripts.

## 3 SSTF: a novel test automation framework

As stated in previous sections, 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 recognised by analyzing software UML diagrams. Test cases can be generated from source code i.e., on the basis of syntax; or from UML diagrams i.e., 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. Keeping the above factors in mind, a new automated test generation framework named semantics and syntax test generation automation framework (SSTF) is proposed.

### *3.1 Overview of SSTF*

During the design of the framework, attention is given to 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.

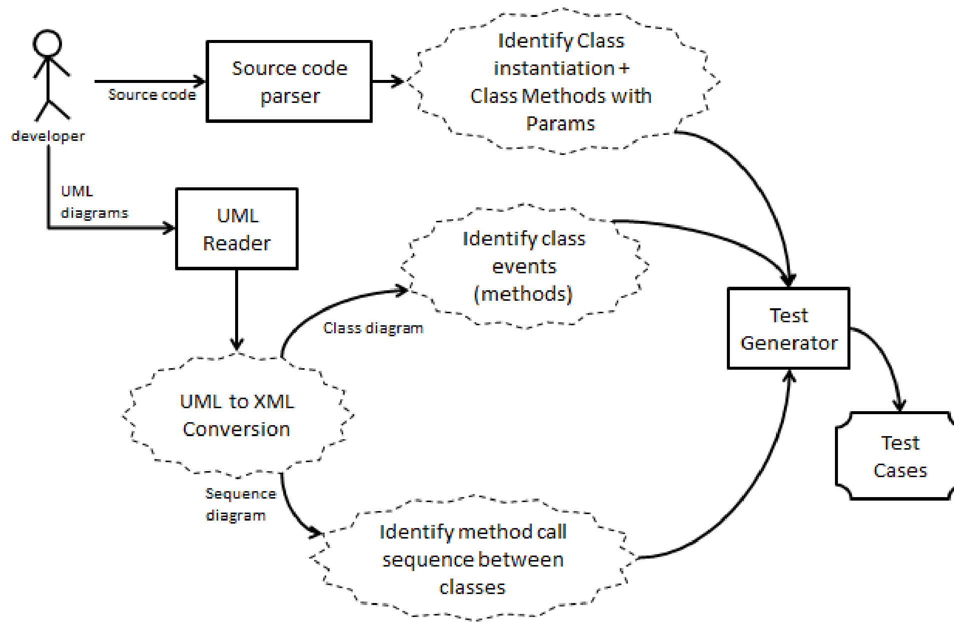
The top-level overview of the proposed architecture is shown in Figure 1. The architecture is divided into three sections as the whole framework stands on three core tasks. The sections are:

- source code parser
- UML reader
- test generator.

The source code parser is designed by highlighting the fact that it will identify the software syntax by classifying object construction and method call structure with parameters. The next component is UR 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 identify the software semantics as shown in Figure 1. 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.

**Figure 1** Top level view of SSTF




## 3.2 Architecture of SSTF

The architecture of the proposed framework is presented in Figure 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

- input layer
- service layer
- test run layer.

Each layer has different responsibilities such as input processing, test generation, etc. The responsibilities of the layers and the components of those are shown in Table 1 along with the acronyms of the components. The input layer is the door, through which the users interact with the framework. It consists of three components – UR, XC and SR (Figure 2). The second layer of the framework is the Service Layer. It is the main layer as all the major computations are performed here. There are six major components in it and two more supporting components for associating those. The last layer is the test run layer,

which is responsible for running the generated test cases. It has three components for performing this task; manual assert statement inserter (MASI) or test editor, test script compiler (TSC) and test script runner (TSR) (Figure 2).

**Figure 2** The component stack of SSTF

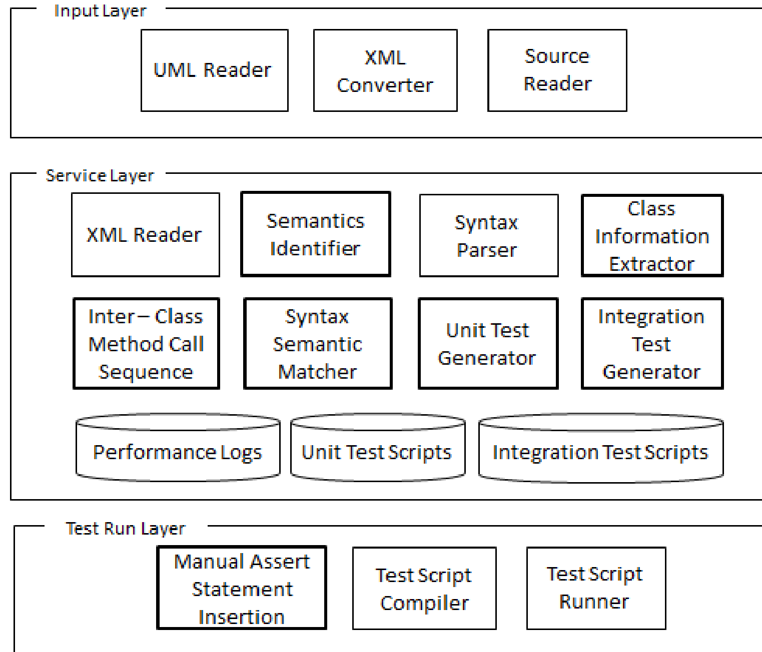



**Table 1** Responsibilities of components of each layer

| *Name* | *Responsibility* |
|---|---|
| Input layer | Input processing |
| UML reader (UR) | Read UML diagrams |
| XML converter (XC) | Converts UMLs to XML files |
| Source reader (SR) | Read source code files from project source location |
| Service layer | Generate test scripts |
| XML reader (XR) | Parse XMLs |
| Semantic identifier (SI) | Extract structured information from parsed XMLs |
| Class information extractor (CIE) | Extract class specific information (from class XML) |
| Inter-class method call sequence (IMCS) | Extract class interaction information (from sequence XML) |
| Syntax parser (SP) | Parse source code |
| Syntax semantic matcher (SSM) | Validation by matching information of both sources (syntactic, semantic) |
| Unit test generator (UTG) | Generate unit test scripts |
| Integration test generator (ITG) | Generate integration test scripts |

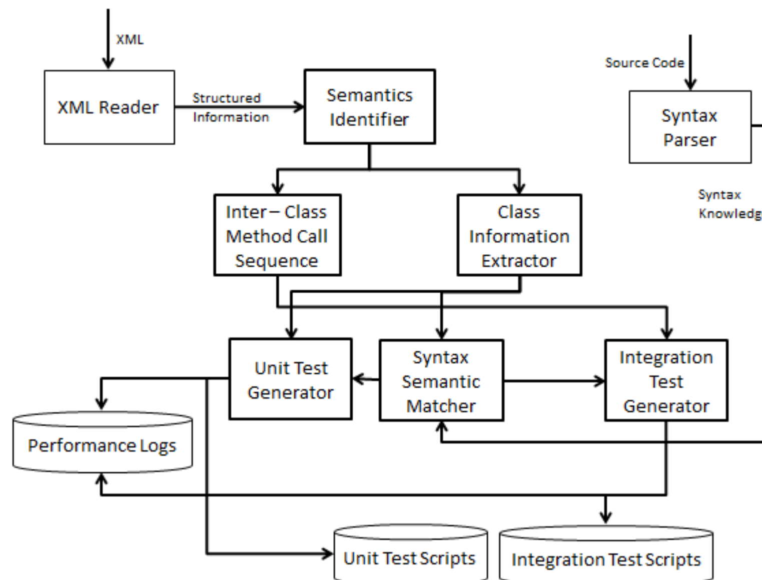
**Table 1** Responsibilities of components of each layer (continued)

| *Name* | *Responsibility* |
|---|---|
| Test run layer | Execute test scripts |
| Manual assert statement inserter (MASI) | Edit test scripts to add assert statements |
| Test script compiler (TSC) | Compile scripts |
| Test script runner (TSR) | Run scripts |

As mentioned earlier, the first layer is composed of three components. UR takes UML diagrams and forwards those to XC.XC converts the diagrams to XML for getting a program readable format (examples of the XMLs can be seen in Figure 6). The third component of this layer is SR which takes the location of the project source and reads the source files.

The six major components of the second layer are mostly responsible for the generation of test cases. The components are the semantics identifier (SI), class information extractor (CIE), inter-class method call sequence (IMCS), syntax semantic matcher (SSM), unit test generator (UTG), and integration test generator (ITG) (as in Figure 2). Figure 3 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 (XR) and syntax parser (SP). XR receives files from XC and reads the information hidden there. SP gets the source code from SR and takes the syntactic information of the software. Finally information produced by these two components is used by the mentioned major components of the layer as shown in Figure 3.

**Figure 3** The interaction among the components of the second layer




CIE and IMCS can be entitled as helpers of SI. These helpers support it for the identification of the software semantics by categorising the tasks to be done. CIE 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. IMCS 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.

SSM is a significant component of this framework that identifies whether the given syntactic and semantic information matched or not. Mismatch can happen because of backdated UMLs or incomplete source code. SSM identifies conflicts between these two information sources and minimises its effect on test generation. For this, names of the classes and names of the methods inside those classes are compared between the class UML and the source code, and the matched portions of those are returned. The reason behind putting this component in action is to ensure that unwanted portions of source code or UMLs are not affecting the generated test cases. SSM is configurable which lets the testers decide to what extent (no matching, only class, both class and method), they want to match syntax and semantics.

UTG and ITG are the most important components. These 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 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 IMCS component. Thus the method information along with call sequence can lead to successful generation of integration test scripts.

For generating unit test scripts, the required information are – name of the class to be tested, class constructor calling syntax, name of the methods to be tested, method calling syntax along with parameter list, desired package and required imports for the test script. First, the class names are are extracted after matching the class diagram class names with the source code. The unit test script class name is given to be the class name by adding ‘Test’ after it. For example, if the class ‘Hello’ is to be tested, the test class name will be ‘HelloTest’. Then the class constructor syntax is extracted from the source code. The constructor parameter types are identified and according to those, the class constructor is called in the script using dummy parameter values. After that, the methods to be tested are extracted from the source code. The method names are gathered along with the parameter types. In the test script, the test methods are created using the names of the methods after adding ‘Test’ behind those. Inside the test methods, the tested class methods are similarly called with dummy parameter values as the constructor. The required imports are identified from the class to be tested and the parameter types. The package name is obtained from the inputted test script class path.

**Algorithm 1** Unit test generation

| | | |
|---|---|---|
| 1 | **procedure** GenerateUnitTests(*syn*, *sem*) | ▷ *syn*: Instance of Syntax, *sem*: Instance of Semantic |
| 2 |     **if** *Config.compare* = *true* **then** | |
| 3 |         *data* ← GetMatched(*syn*, *sem*) | |

```
4        else
5            data ← syn
6        end if
7        unitTests ← []
8        for Each class cl in syn.classes do
9        if cl.isInterface = true or cl.isAbstractClass = true then
10           continue
11         end if
12         unitTest:className ← cl.className + "Test"
13         unitTest.packageName ← Config.packageName
14         unitTest.testPath ← Config.path
15         unitTest.imports ← GetRequiredImports(cl)
16         unitTest.classConstructionString ← GetClassInitialisationString(cl)
17         for Each method me in cl.methods do
18               testMethod.methodName ← me.methodName + "Test"
19               testMethod:parameters ← me.parameters
20               unitTest.testMethods.push(testMethod)
21         end for
22         unitTests.push(unitTest)
23       end for
24       CreateUnitTestScripts(unitTests))
25   end procedure
```

The unit test generation algorithm is shown in Algorithm 1. It takes input as follows: class information extracted from the class diagrams (as semantics) and syntax information from the source code. These information are refined by SSM first as in line 3. The matched methods of the matched classes are selected for the unit test generation (nested loops in Algorithm 1 line 8 and 17). The syntax information provides the syntax of these selected methods (Algorithm 1 line 18–20). The parametrised 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.

For generating integration script, the needed information are – names of the classes involved in the integration test, calling syntax of the class constructors, names of the methods to be called, the sequence of method calls, method calling syntax, package and import list. First of all, the integration test is revealed from the sequence diagram. The name of the sequence diagram is taken as the test class name by adding 'Test' next to it. The test method is also named the same. The classes involved in the integration test are identified from the *lifelines* of the sequence diagram. The constructor calling syntax are extracted from the source code. The parameter types of the constructors are discovered from the code, and the constructors are called by providing dummy values of those types. Then, the methods to be called are identified from the *messages* between the *lifelines* in the sequence diagram along with their sequence of call. The method calling syntax are collected from the source code. Then according to the sequence of call, the methods are

called by using the source code syntax. Dummy parameter values are used for the identified parameter types of the methods. The import list is collected from required classes and the required parameter types.

The integration test generation algorithm is shown in Algorithm 2. It uses the class interaction information hidden in the sequence diagrams (semantics) along with the syntax information and generates integration test scripts. IMCS component reveals the method calls between classes, that are concealed in the sequence diagrams. This method call information works as input in the generation of integration tests. For each sequence diagram, the events (methods) of the diagrams are called as per the method call syntax (Algorithm 2 nested loop from line 3 to 20). Similar to the unit test scripts, these integration test scripts are also stored to be run later by the user.

**Algorithm 2** Integration test generation

| | | |
|---|---|---|
| 1 | **procedure** GenerateIntegrationTests(*syn*, *sem*) | ▷ *syn*: Instance of Syntax, *sem*: Instance of Semantic |
| 2 | *integrationTests* ← [] | |
| 3 | **for** Each sequence *seq* in *sem.sequenceDiagrams* **do** | |
| 4 | *integrationTest.className* ← *seq.label* + "*Test*" | |
| 5 | *integrationTest.packageName* ← *Config.packageName* | |
| 6 | *integrationTest.testPath* ← *Config.path* | |
| 7 | *integrationTest.testMethod:methodName* ← *seq.label* + "*Test*" | |
| 8 | *matches* ← Get matched classes syntax from *sem.actors* | |
| 9 | *integrationTest:imports* ← GetRequiredImports(*matches*) | |
| 10 | *integrationTest.listOfIntegratedClass Constructors* ← GetClassInitialisationString(*matches*) | |
| 11 | **for** Each event *ev* in *seq.events* **do** | |
| 12 | *calledClass* ← Get called actor class syntax from *ev* | |
| 13 | *calledMethod* ← Get called method syntax from *calledClass* and *ev.label* | |
| 14 | *testMethod.methodName* ← *calledMethod.methodName* + "*Test*" | |
| 15 | *testMethod.parameters* ← *calledMethod.parameters* | |
| 16 | *integrationTest.methodCallSequence.push*(*testMethod*) | |
| 17 | *integrationTest.methodCallClassVar.push*(*calledClass*) | |
| 18 | **end for** | |
| 19 | *integrationTests.push*(*integrationTest*) | |
| 20 | **end for** | |
| 21 | *CreateIntegrationTestScripts*(*integrationTests*)) | |
| 22 | **end procedure** | |

The last layer is in charge of running the generated test cases. MASI component of the layer needs human interaction for the supplement of assert statements inside the generated test scripts. TSC and TSR work together to run the test cases and detect the software faults (e.g., JUnit for Java projects).

## 4 A case study on project 'ObserverPatternExample'

For an assessment of the competency of the approach, SSTF was used on a Java project illustrating observer pattern. The observer pattern is a 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 (Observer Pattern|Object Oriented Design, 2014). The example project has a class called person as observer class, a class called Product as subject class, and the observer and subject interfaces (SampleTestProjects-Java-GitHub, 2014).

### 4.1 Analysis of source code

Figure 4(a) and 4(b) show snapshots of the source code of the person and product class accordingly. SR takes this project source location as input and reads the Java files to be processed later by SP in second layer. SP 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.

**Figure 4** Source code example, (a) Class: person (b) Class: product (see online version for colours)

```
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);
    }

    public void unUsedMethod() {
        System.out.println("I am garbage");
    }
}
```

(a)

**Figure 4** Source code example, (a) Class: person (b) Class: product (continued) (see online version for colours)

```
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);
    }
```

(b)

For example, for Figure 4(a), class name is *Person*; it has one constructor taking *personName* (type string) as parameter; the methods are *getPersonName* (no parameter), *setPersonName* (one string parameter), *update* (one string parameter), *unUsedMethod* (no parameter). Similarly, in Figure 4(b), class name is *Product*. The constructor takes three string parameters, *productName*, *productType* and availability. It contains three methods named *notifyObservers* (no parameter), *registerObserver* (one *Observer* class type parameter) and *removeObserver* (one *Observer* class type parameter).

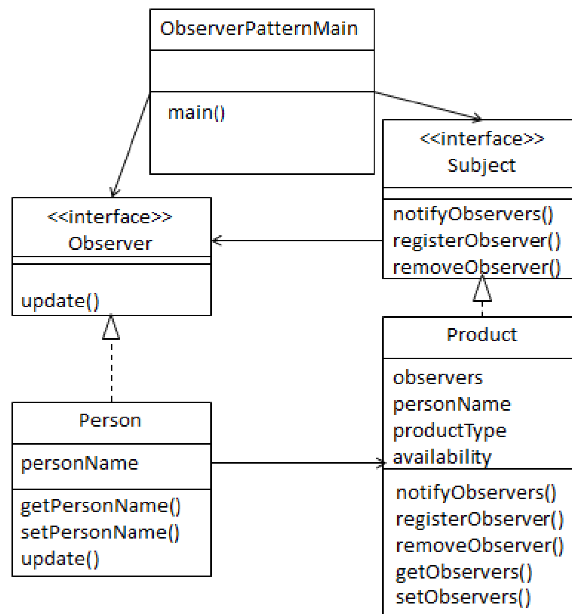
## *4.2 Getting XMLs from UMLs*

The project UML diagrams are inputted in UR (components are shown in Table 1). These are then converted to XML using XC component. Enterprise architect is used for this conversion purpose (Enterprise Architect – UML Design Tools and UML CASE Tools for Software Development, 2014). Figure 5 shows the class and sequence diagrams, and a portion of the produced XMLs are illustrated in Figure 6. The XML portion of the class diagram is for *Person*. The <owedAttribute> tags of the class XML refer class variables and the <owedOperation> tags refer class methods.

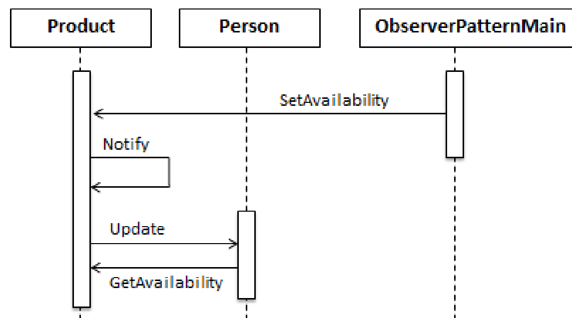
In Figure 5(a), three classes and two interfaces can be seen to be connected with each other. The classes are – *ObserverPatternMain*, *Person* and *Product*, and the interfaces

are – *Observer* and *Subject*. The attributes and methods can also be identified from the mentioned class diagram. A portion of the generated XML from this class diagram is shown in Figure 6(a), where only the Person class is depicted. Here, *Person* has one attribute stored in the <ownedAttribute> tag – *personName* and three <ownedOperation>s – *getPersonName*, *setPersonName*, and *update*. In the sequence diagram of the sample project, shown in Figure 5(b), there are three *lifelines* – *Product*, *Person* and *ObserverPatternMain*. The *messages* among these *lifelines* are *SetAvailability* (from *ObserverPatternMain* to *Product)*, *Notify* (from *Product* to *Product*), *Update* (from *Product* to *Person*) and *GetAvailability* (from *Person* to *Product*). The generated sequence XML portion in Figure 6(b) shows one method call – *GetAvailabilty*. All the method calls are contained in <connector> tags, and has <source> (*Person*) and <target> tags (*Product*).

**Figure 5** The UML diagrams of the sample project, (a) Class diagram (b) Sequence diagram




(a)




(b)

### *4.3 Analysis of XMLs*

The second layer takes the source and the produced XML as input. XR component of the layer takes the XMLs and sends those to appropriate analyzer. CIE which is the class diagram analyzer, analyzes XML of the class diagram and identifies the class methods, variables as well as class association.

As mentioned already, the <owedAttribute> tags refer to class variables and the <owedOperation>s refer class methods. The values of these are identified by analyzing the tags. For Figure 6(a), *personName* is variable and *getPersonName*, *setPersonName*, update are methods. IMCS is the sequence diagram analyzer and extracts sequence diagram information similarly from their XML [Figure 6(b)] by parsing the appropriate tags (<connector>, <source>, <target>, etc.) in it.

### *4.4 Matching syntax with semantics*

These gathered information are then compared in SSM to check whether there is any inconsistency between the syntax and semantic information, so that those inconsistencies can be avoided while generating test scripts. For assessing the efficiency of this component, some extra classes and methods were added in source [e.g., *unUsedMethod* in source code as show in Figure 4(a)].Some extra classes were also added in the class diagram. The matched classes and methods were identified by checking class and method names. Remaining classes and methods of both source code and diagram were considered as mismatched elements and were eliminated by the component.

*unUsedMethod* was eliminated from Person as the method does not match with any methods of *Person* in class diagram [Figure 5(a)]. So the remaining methods are *getPersonName*, *setPersonName* and update.

**Figure 6** Portions of generated XMLs, (a) Class XML (partial: Person class) (b) Sequence XML (partial: GetAvailaility call)

```
<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)

**Figure 6** Portions of generated XMLs, (a) Class XML (partial: Person class) (b) Sequence XML (partial: GetAvailaility call) (continued)


```
<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>
</connector>
```


(b)

## 4.5 Generation of unit test

The semantic class information along with the class syntax are inputted in UTG. 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.

A sample unit test suite snapshot (for *Product class*) is shown in Figure 7(a). The *setUp* stub contains instantiation of *Product* using syntax got from Figure 4(b). In Figure 4(b), it is seen that the *Product* constructor needs three parameters: *productName*, *productType* and *availability*. These three parameters accept string values. Thus, dummy string parameters are passed in the constructors for instantiating the *Product* object. For all the methods of *Product*, test methods are generated where the actual methods are called based on the stored syntax information, and three of those are shown in the snapshot [Figure 7(a)]. The *registerObserver* method takes a parameter of type *Observer* as input [as shown in Figure 4(b)]. Thus, mock *Observer* object is created and passed as parameter in the method call. Similarly, in the test stub of *removeObserver*, mock *Observer* variable is passed as parameter of the method call. On the other hand, *setProductName* method accepts a string parameter. In the test stub, dummy string value ‘test’ is passed as parameter.

**Figure 7** Test cases example, (a) Unit test (b) Integration test (see online version for colours)

```
@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");
}
```

(a)

```
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();
}
```

(b)

### 4.6 Generation of integration test

For the generation of integration test cases, sequence diagrams contain the most significant information. The method call sequence obtained 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.

Figure 7(b) shows the snapshot of integration test generated for Figure 6(b). In the *setUp* stub, all the *lifelines* classes of the sequence diagram (*Product*, *Person*, *ObserverPatternMain*) are instantiated as those are the interacting active classes. Then in the test method, those class methods are called in order of the sequence [as Figure 6(b) *SetAvailability*, *Notify*, *Update*, *GetAvailailty*]. The syntax of the instantiation and method call are obtained similarly as the unit test generation from source code syntax.

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. The use of sequence diagram for generation of integration test reduces efforts for extracting method call sequence from source code.

## 5 Implementation and result analysis

This section aims to experimentally evaluate the performance of SSTF. A prototype of SSTF has been implemented in Java. The execution time, number of generated tests, number of useful (runnable) test cases were selected as 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 syntactic approach named as Fusion (Pezze et al., 2013) and semantic approach of Sharma and Biswas (2014). The results were analyzed to get an estimation about the comparative usefulness of SSTF.

### 5.1 Environmental setup

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

- Eclipse Kepler (4.3): Java IDE for SSTF implementation (Eclipse.org – Kepler Simultaneous Release: Highlights, 2014).
- Javaparser version-1.0.8: Java library for source code parsing (Javaparser, 2014).
- Enterprise architect (EA): UML editor and XC (Enterprise Architect – UML Design Tools and UML CASE tools for software development, 2014).
- Eclipse Plugin JUnit 4: test script compiler and runner (JUnit, 2014).

The prototype of SSTF (SSTF-GitHub, 2014) 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 effect take place.

The experiments were performed on the Desktop configuration of 2.20 GHz Intel Core 2 Duo Processor having 6GB of RAM, Windows 8 OS and Java 7; change in this configuration might affect the experimented response times. For the assessment of SSTF, four different projects (SampleTestProjects-Java-GitHub, 2014) were selected along with their UMLs. These projects are the student projects, developed for different courses in Institute of Information Technology, University of Dhaka. Table 2 shows the projects with its attributes – line of code (LOC), number of classes in class diagram and number of sequence diagrams. Class diagrams were not provided for two projects (CWR and CFG), to test whether the framework can still generate unit test scripts depending on the source code. However, for generating integration test scripts, sequence diagrams must be provided.

**Table 2** Experimented projects

| *Project name* | *LOC* | *No. of classes in class diagram* | *No. of sequence diagrams* |
|---|---|---|---|
| Observer pattern example (OPE) | 141 | 5 | 1 |
| Calendar with reminder (CWR) | 1,378 | Not Provided | 1 |
| Connect four game (CFG) | 1,769 | Not Provided | 4 |
| Calculator | 2,205 | 44 | 2 |

Prearrangement, needed to be done before running SSTF on the sample project set, are as follows. 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 (Software Design Tools for Agile Teams, with UML, BPMN and More, 2014). The projects run properties need to be included in a configuration file, where the project path, XML path, test package, and tester's decisions about comparing syntax-semantic classes and methods are specified. The configurations of SSM, about comparing syntax-semantic classes and methods, can be given as *no matching, only class matching or both class and method matching*.

## *5.2 Comparative analysis*

For the comparative analysis, SSTF, the selected semantic approach and the syntactic one (named as, Fusion) were executed on the projects mentioned above and the results are compared. 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 the ratio of runnable tests in the generated tests is measured to be the effectiveness. Finally, the satisfaction is 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 2. SSTF was run 10 times on the projects and the average time was computed. Table 3 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 0.48, 0.56, 0.58 and 1.03 seconds to analyze and generate test cases for the four projects in the set. Moreover, it generates 15, 43, 20 and 87 test cases accordingly for these projects in the elapsed time. Similarly, Fusion was executed 10 times on the sample project set and the average time was taken as the final execution time. As the number of generated test cases varies for SSTF and Fusion, time per test is calculated for the time comparison of the two approaches. It is found that for each of the projects, Fusion takes more time than SSTF to generate a test script. Thus, Table 4 shows that required time per test generation in SSTF is less than Fusion for all the projects in the selected dataset.

**Table 3** Test cases generated with SSTF and average execution time on the desktop configuration

| *Project* | *No. of generated unit tests* | *No. of generated integration tests* | *Avg. execution time (sec)* |
|---|---|---|---|
| OPE | 15 | 1 | 0.4756 |
| CWR | 43 | 1 | 0.5585 |
| CFG | 30 | 4 | 0.5788 |
| Calculator | 87 | 2 | 1.0344 |

**Table 4** Comparison of execution time between SSTF and fusion

| *Project* | *Generated tests* | | *Avg. time (sec)* | | *Time per test* | |
|---|---|---|---|---|---|---|
| | *SSTF* | *Fusion* | *SSTF* | *Fusion* | *SSTF* | *Fusion* |
| OPE | 16 | 10 | 0.4756 | 0.3 | 0.0297 | 0.03 |
| CWR | 44 | 55 | 0.5585 | 2.8 | 0.01269 | 0.0509 |
| CFG | 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.

The semantic approach of Sharma and Biswas (2014) cannot be compared to SSTF in terms of execution time. Because, the approach does not generate test scripts. It only defines test cases in the early development phase, that can be used to generate test scripts later.

### 5.2.2 Effectiveness

The effectiveness is measured by the proportion of generated useful test scripts. However, the way to measure the usefulness of the test scripts is different for different approaches because of the dissimilar procedures and motivations. For example, SSTF

generates unit and integration test scripts using UML and source code. On the other hand, Fusion generates integration tests using already generated unit tests and the software source code. Thus, the inputted unit test scripts is also an parameter in the measure of generated test script effectiveness for Fusion. Again, the semantic approach (Sharma and Biswas, 2014) takes courteous logic as input and generates test cases (not test scripts). Thus, the effectiveness of these test cases cannot be measured similarly as the generated test scripts by the other two approaches.

For SSTF, percentage of runnable test scripts are considered as effectiveness. On the other hand, as unit tests are inputted in Fusion, the utilisation of those unit tests are also a factor along with the runnability of the generated scripts. Thus, the measurement of effectiveness for Fusion is calculated as the success rate using the following equation:

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

where

$T_{compilable}$ number of compilable scripts

$T_{generated}$ number of generated scripts

$T_{skipped}$ number of provided unit tests for which no test cases was created.

The semantic approach does not generate runnable scripts, and so runnability cannot be a measurement of it. In this approach, test cases are only determined that are later used to create test scripts manually. Thus, no compilable scripts are generated using this approach. As a result, the effectiveness of the test cases could not be measured due to the absence of test scripts.

Table 5 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 which caused a compilation error (Interfaces cannot be instantiated). 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.

**Table 5** Test cases generated with SSTF and ratio of runnable tests

| *Project* | *No. of generated test cases* | *No. of compilable and runnable tests* | *Ratio (%)* |
|---|---|---|---|
| OPE | 16 | 16 | 100% |
| CWR | 44 | 44 | 100% |
| CFG | 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 (Pezze et al., 2013), while the remaining 40% needs manual modification to become executable. However, for the selected project set the success rate was 50%, 80.88%, 77.78% and 86.91%. Table 6 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 considered as test generation failure.

Now, Table 7 shows the number of generated test cases using the semantic approach by Sharma and Biswas (2014). As, these test cases are not provided in runnable test script form, the compilability ratio could not be measured. Thus, the success rate comparison with the other two approaches could not be performed. However, the number of the generated test cases indicates that the approach has skipped the generation of many test cases. While for project OPE, SSTF generated 16 test cases, this approach could identify 10. For project CWR, 21 cases were produced, while SSTF provided 44 cases for the same project. This approach could found 16 test cases for project CFG, while SSTF generated 31. Most of the test cases are skipped for project Calculator; while, SSTF generated 89 test cases for the project, this approach generated only 10.

**Table 6** Test cases generated with fusion and ratio of runnable tests

| *Project* | *No. of generated test cases* | *No. of compilable test cases* | *No. of skipped tests* | *Success rate (%)* |
|---|---|---|---|---|
| OPE | 10 | 5 | - | 50% |
| CWR | 55 | 55 | 13 | 80.88% |
| CFG | 35 | 35 | 10 | 77.78% |
| Calculator | 180 | 166 | 11 | 86.91% |

**Table 7** Test cases generated with the semantic approach

| *Project* | *No. of generated test cases* | *No. of compilable and runnable tests* | *Ratio (%)* |
|---|---|---|---|
| OPE | 12 | undefined | undefined |
| CWR | 21 | undefined | undefined |
| CFG | 16 | undefined | undefined |
| Calculator | 10 | undefined | undefined |

### *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 encouraging 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 methods in all classes (only public methods can be tested by a test case, as private methods cannot even be accessed by other classes except the parent class), 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 synchronised with it. Thus, achieving test coverage seems to be more appropriate in test data generation than test script generation. However, the measurement of test coverage is not given priority in this research as test coverage is loosely correlated with test suite effectiveness (Inozemtseva and Holmes, 2014).

About the configurable SSM component of SSTF, the experiments are run on all the different configurations. For example, for project *OPE*, *SSM* was configured for *only class matching*. For project *calculator*, *both class and method matching* was applied. For *CWR* and *CFG*, *no matching* was used as the configuration. These configuration files can be viewed in SampleTestProjects-Java-GitHub (2014). For these small projects the configurations did not have much effects on the test generation, because, the syntactic and semantic differences in these projects were not significant. There was only a small difference in the number of generated test scripts for different configurations on a project. Thus, it was not reported on this paper. On the other hand, it is also not the focus of this work. It can be a future scope to experiment different SSM configurations on a diverse dataset.

### *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 do 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 do not synchronise 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 tests while software source code and unit test scripts are available. 60% of the Fusion generated test scripts are runnable leaving the remaining 40% unusable (Pezze et al., 2013). The reason behind generation of these 40% unsuccessful test scripts is that, syntax parsing is a complex task making it difficult to find software integration information from inside code. This task has been made easier by the incorporation of sequence diagram with software syntax.

The semantic approach (Sharma and Biswas, 2014) cannot generate any runnable test scripts. This is because, it has no syntax information for producing the scripts. Also, as this approach do not use the source code but the initially created specification, it do not have the final reflexion of the software. This is why, it generates a smaller number of test cases, not identifying all the software components to test. SSTF merges software syntax with semantics, allowing it to generate runnable test scripts for all the software components.

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 synchronised together before the test generation.

## 6 Threat to validity

An *internal validity* threat of this research is the simple implementation of the SSM component. As mentioned in the architecture of SSTF, unit test generator (UTG) and and integration test generator (ITG) use the syntax from source code and semantics from diagrams for test generation. In the process, syntax semantic matcher (SSM) tries to reduce the distance of a backdated SRS with source code so that it does not affect the generated tests. However, as SSM was not the focus of this work, the implementation is a simple one. Here, only names of the classes and names of the methods inside those classes are compared between the class UML and the source code. The implementation can be improved by providing a more complex solution, for example, comparison of method behaviour (using activity diagram and source code control flow); comparison of different code metrics like cyclomatic complexity, cohesion, coupling, halstead complexity, etc; using code search methods with amalgamation of useful UMLs. The proposed test generation technique in this paper is not dependent on the SSM component. This component only tries to polish the syntactic and semantic inputs before generating tests from those. This component can be another area of research in future for supporting different works such as test script generation, test data generation, test case prioritisation, performance testing (Gias and Sakib, 2014), etc. In this work, SSM was not our major contribution, but only a part to support our contribution of test generation.

An *external validity* threat is that SSTF has been applied on small in-house classroom projects. Experimentation on industrial and large projects could not be performed due to the unavailability of design documents. Also, experimentation of the configurable SSM component has not been included in this paper due to being insignificant and out of the scope of this work.

## 7 Conclusions

Unlike conventional test generation approaches, this research proposes to incorporate software syntax and semantics. A framework named SSTF is proposed for this purpose, which uses UMLs for obtaining software internal structure and interactions, and source code for deriving test scripts' body.

The framework consists of three layers for the test script generation – input layer, service layer and test run layer. The input layer processes the user inputs (UML diagrams and source code) using three components (UR, XC and SR). The service layer has eight components for extraction of software syntax, semantic; and their incorporation in test generation. The test run layer is responsible for providing the test script editor, compiler and runner.

A step-by-step case study evaluates the competence of the approach to construct proper tests. The experiments conducted on sample projects show that the elapsed time of SSTF is quite low (0.66 seconds on average for the sample project set) and 99.72% scripts are runnable which is an immense improvement compared to the two other semantic and syntactic test generation tools.

The future challenge lies in the integration of proper test data with the generated test scripts of SSTF for achieving test coverage.

## References

Arcuri, A., Fraser, G. and Galeotti, J.P. (2014) 'Automated unit test generation for classes with environment dependencies', in *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering* (*ASE*), pp.79–90.

Beizer, B. (1990) *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, New York.

Cartaxo, E.G., Neto, F.G.O. and Machado, P.D.L. (2007) '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.

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

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

Deursen, A.V., Klint, P. and Visser, J. (2000) 'Domain-specific languages: an annotated bibliography', *ACM Sigplan Notices*, Vol. 35, No. 6, pp.26–36.

Diaz, E., Tuya, J. and Blanco, R. (2003) 'Amodular tool for automated coverage in software testing', in *Proc. of the 11th IEEE Annual InternationalWorkshop on Software Technology and Engineering Practice* (*STEP*), pp.241–246.

Duclos, E., Digabel, S.L., Gueheneuc, Y. and Adams, B. (2013) '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.

Dustin, E., Garrett, T. and Gauf, B. (2009) *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, p.368, Addison-Wesley Professional.

Eclipse.org – Kepler Simultaneous Release: Highlights (2014) [online] http://eclipse.org/kepler/ (accessed 2014).

Enoiu, E.P., Sundmark, D. and Pettersson, P. (2013) '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.

Enterprise Architect – UML Design Tools and UML CASE Tools for Software Development (2014) [online] http://www.sparxsystems.com/products/ea/ (accessed 2014).

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

Fraser, G. and Arcuri, A. (2011) '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* (*FSE*), pp.416–419.

Gias, A. and Sakib, K. (2014) 'An adaptive bayesian approach for url selection to test performance of large scale web-based systems', in *Companion Proc. of the 36th International Conference on Software Engineering* (*ICSE*), pp.608–609.

Glover, F. (1997) *Tabu Search*, 382pp., Kluwer Academic Publishers Norwell, MA, USA.

Godefroid, P., Klarlund, N. and Sen, K. (2005) 'DART: directed automated random testing', in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*), pp.213–223.

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

Javaparser (2014) [online] https://code.google.com/p/javaparser/ (accessed 2014).

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

JUnit (2014) [online] http://junit.org/ (accessed 2014).

Khatun, A. and Sakib, K. (2016) '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.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J and Irwin, J. (1997) *Aspect-Oriented Programming*, Springer.

Lamsweerde, A.V. (2001) *Building Formal Requirements Models for Reliable Software*, Springer Berlin Heidelberg.

Leitner, A., Ciupa, I., Meyer, B. and Howard, M. (2007) 'Reconciling manual and automated testing: the auto test experience', *Proc. of the 40th Annual Hawaii International Conference on System Sciences* (*HICSS*), p.261a.

Martia, R., Lagunab, M. and Gloverb, F. (2006) 'Principles of scatter search', *European Journal of Operational Research*, Vol. 169, No. 2, pp.359–372.

Nahar, N. and Sakib, K. (2014) '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.

Nebut, C., Fleurey, F., Traon, Y.L. and Jezequel, J. (2006) 'Automatic test generation: a use case driven approach', *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp.2–8.

Observer Pattern|Object Oriented Design (2014) [online] http://www.oodesign.com/observer-pattern.html (accessed 2014).

Pacheco, C., Lahiri, S.K., Ernst, M.D., and Ball, T. (2007) 'Feedback-directed random test generation', in *Proc. of the 29th International Conference on Software Engineering* (*ICSE*), pp.75–84.

Pezze, M., Rubinov, K. and Wuttke, J. (2013) '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.

Ramler, R., Weippl, E., Winterer, M., Schwinger, W. and Altmann, J. (2002) 'Aquality-driven approach to web testing', in *Proc. of the Iberoamerican Conference on Web Engineering* (*ICWE*), Vol. 2, pp.81–95.

SampleTestProjects-Java-GitHub (2014) [online] https://github.com/NadiaIT/SampleTestProjects-Java (accessed 2014).

Sharma, M. and Chandra, B.S. (2010) '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.

Sharma, R. and Biswas, K.K. (2014) 'Automated generation of test cases from logical specification of software requirements', in *Proc. of the 2014 International Conference on Evaluation of Novel Approaches to Software Engineering* (*ENASE*), pp.1–8.

Singi, K., Era, D. and Kaulgud, V. (2015) 'Model-based approach for automated test case generation from visual requirement specifications', in *Proc. Of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*), pp.1–6.

Software Design Tools for Agile Teams, with UML, BPMN and More (2014) [online] http://www.visual-paradigm.com/ (accessed 2014).

SSTF-GitHub (2014) [online] https://github.com/NadiaIT/SSTF (accessed 2014).

Taneja, K. and Xie, T. (2008) 'DiffGen: automated regression unit-test generation', in *Proc. of the 23rd IEEE International Conference on Automated Software Engineering* (*ASE*), pp.407–410.

UPPAAL (2014) [online] http://www.uppaal.org/ (accessed 2014).

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