SSTF: A Novel Automated Test Generation Framework using Software Semantics and Syntax

Conference Paper · December 2014

DOI: 10.1109/ICCITechn.2014.7073069

2 authors:

Nadia Nahar
University of Dhaka
5 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Kazi Sakib
University of Dhaka
40 PUBLICATIONS **52** CITATIONS
SEE PROFILE

SSTF: A Novel Automated Test Generation Framework using Software Semantics and Syntax

Nadia Nahar* and Kazi Sakib†
Institute of Information Technology, University of Dhaka, Bangladesh
*nadia.nahar.iit@gmail.com, †sakib@univdhaka.edu

***Abstract*—Test Automation saves time and cost by digitizing the process of test generation and execution. The automated test generation techniques in the literature do not always produce effective and compilable test cases. A test generation framework is proposed in this paper which uses the information extracted from UML diagrams and source code. The three layer architecture of the framework is responsible for this generation task. The first layer processes the user inputs i.e the UMLs as XMLs and the source code as source classes, which are used by the second layer. This layer identifies the application semantic from XMLs and extracts syntax from source. It combines this extracted information together and generates unit and integration test cases. This incorporation of syntax and semantics should make the generated tests less erroneous as it creates a better understanding of the application before the test construction. These two directional information collection should also mitigate the negative effects of inconsistent UMLs or source code on the test suites. Moreover, the generation of both unit and integration test case may increase the test coverage. A case study, conducted on a sample java project, assessed the framework competence and has been successful to construct test scripts.**

***Keywords*—software testing, automatic test generation, unit testing, integration testing**

I. Introduction

Automated testing is a process where software is tested without any manual input, analysis or evaluation. Automating the testing process may create an impact on software development cost, since testing consumes at least 50% of the total costs [1]. A tester, considered as a critical analyzer, has profound knowledge on software semantics i.e. the requirements and the syntax i.e. 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. However, understanding the program semantically and incorporating syntax with it without human interaction may either miss some coverage area or produce redundant test cases.

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. 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. 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. However, as SRS is created in the early development stage, the diagrams are often backdated and do not match the software code segment 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 Mauro Pezze et al. [2], 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 [3]. This tool automates instrumentation process to decrease testing time and errors due to manual instrumentation, and 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. An automatic test case generation approach using UML activity diagrams was presented by Chen et al. [4]. At the same year, Nebut et al. proposed another new approach for the automation in the context of Object-Oriented (OO) software [5]. Both these papers used semantic approach for the test generation, and thus suffer from the mentioned drawbacks.

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 ex-

traction 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 got from source. Finally, the Test Run Layer has the role to insert assert statements, compile and run tests.
A case study has been conducted here for the initial assessment of the proposed framework. The case study was carried out on a simple java project illustrating a popular design pattern, the observer pattern. The sample project source code containing an observer class i.e. Person class, a subject class i.e. Product class and the Observer-Subject interfaces was written in java. The corresponding UMLs of the sample projects were also built. These source and UMLs were inputted in the User End Layer. The output were some XML data and source classes. These were received by the Service Layer and after processing, the output was the test scripts. These scripts were then successfully run by the Test Run Center.
The proposed framework uses the class, state and sequence diagrams to gather semantic information and extracts syntactic information from the source code. Together this information creates enough understanding of the software internal and external structures. This understanding should make the generated test cases less error-prone and decrease creation of ineffective tests. Thus, the incorporation of syntax with semantics may make the generation of test cases more effective.

II. Related Work

In the literature several automatic test case generation techniques have been proposed. Most of those techniques consider either syntactic [6] or semantic [7] approach. 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.
A prominent research of the syntactic approach proposed a tool for automatic generation of test cases from source code so as to attain branch coverage in software testing [3]. A parser, an instrumenter, and different test case generators (such as Tabu Search, Scatter Search, and Random based) have been combined here for C/C++ programs. The testing time and errors due to manual instrumentation have been decreased by the use of this tool. However, the instrumentation time of the framework is too high which could be improved by considering additional extracted information from UML.
Fix et al. presented the design and implementation of a framework for automated unit test code generation [8]. The framework was developed using XSLT, XML and the C# programming language in .Net managed environment. Also a semi-automatic generation process of unit test code was described here along with the framework. The last product of the framework is some basic unit test code which can be executed into a test run environment such as the Visual Studio IDE or NUnit runner program. However, the generation of these basic test cases can easily be improved to generation of complex test cases by incorporation of semantic information.
An approach to generate complex or integration test cases using simple or unit ones has been proposed by Pezze et al. [2]. 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 application syntax which was used to construct more complex tests focusing on the interaction of the classes. However, 60% of the generated test cases were executable while the remaining 40% could not be compiled. If the semantic information was incorporated with the extracted source information, application behavior could be clearer and the result be improved.
The literature of test generation automation by semantic approach is a rich field. Nebut et al [5]. proposed a new approach for the automation in the context of object-oriented software. A complete and automated chain for test cases was derived from formalized requirements of embedded OO software here. However, here, 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 Chen et al. [4]. The test generation automation here supports some test adequacy criteria for the activity diagrams. It also corresponds to Model Driven Architecture (MDA) [9] and the Model Driven Testing (MDT) [10]. However, the generation of basic test cases in the paper might have 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 [11] and researchers are exploring software model usage to support Verification and Validation (V&V) activities. Javed et al. established a method that uses the model transformation technology of MDA to generate unit test cases by using sequence diagrams [12]. The method was implemented using Eclipse, Tefkat, MOFScript, JUnit and SUnit; and is general for implementation in any other xUnit testing framework. However, the fact that outdated UMLs can generate ineffective test cases had not been considered by the authors.
Regression test generation was also emphasized by some authors. Taneja et al. presented an automated regression unit-test generator called DiffGen for Java programs [13]. Moreover, a black-box testing technique was considered by Sharma et al. to test suite auto-generation [14]. In both cases if UML diagrams were considered and compared along with the source, generated tests could have been better.
These researches address the importance of test generation automation. Although various frameworks have been proposed throughout years, none of these are complete and flawless. Researchers have applied different approaches for improving the results, but complete success could not be attained yet. Incorporation of these approaches can encompass the desired objective, so further research is needed.

III. SSTF: A Novel Test Automation Framework

In this section, a new automated test generation framework is proposed. 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 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.

A. 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.
The top-level overview of the architecture is shown in Fig. 1. The architecture is divided into three sections as the whole framework stands on three core tasks. The sections are: (1) Source Code Parser, (2) UML Reader and (3) Test Generator. The first section, Source Code Parser, is assimilated with the

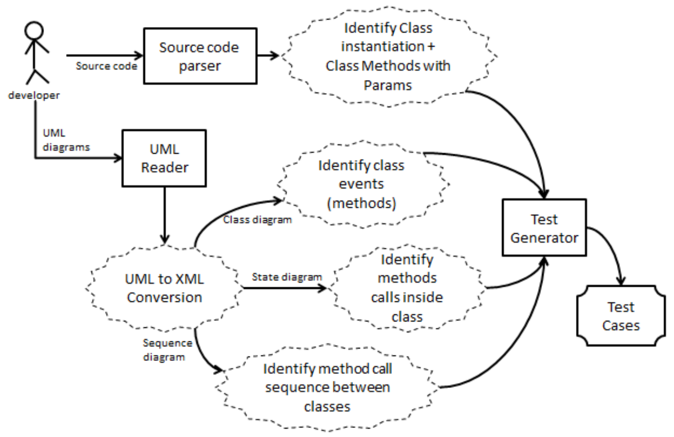

developer
Source code
Source code parser
Identify Class instantiation + Class Methods with Params
UML diagrams
UML Reader
Identify class events (methods)
Class diagram
Test Generator
UML to XML Conversion
State diagram
Identify methods calls inside class
Test Cases
Sequence diagram
Identify method call sequence between classes

Fig. 1: Top level view of SSTF

IDE of the clients. It is designed by highlighting on the fact that it will identify the software syntax by classifying initialization of objects and call sequence of methods with parameters. The next component is UML Reader that works as the semantic identifier. The provided UML diagrams i.e. the class, state 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 state and sequence diagrams accordingly.

B. Architecture of SSTF

The architecture of the framework is presented in Fig. 2. The component stack of the architecture can be represented in two categories. The thick bordered components are the major ones while the thin bordered are the supporting components to those. The framework is separated in three layers: (1) User End Layer, (2) Service Layer and (3) Test Run Center.
Each layer has different responsibilities. The User End Layer is the door, through which the users interact with the framework. The second layer of the framework is the Service Layer. It is the main layer as all the major computations are done here.

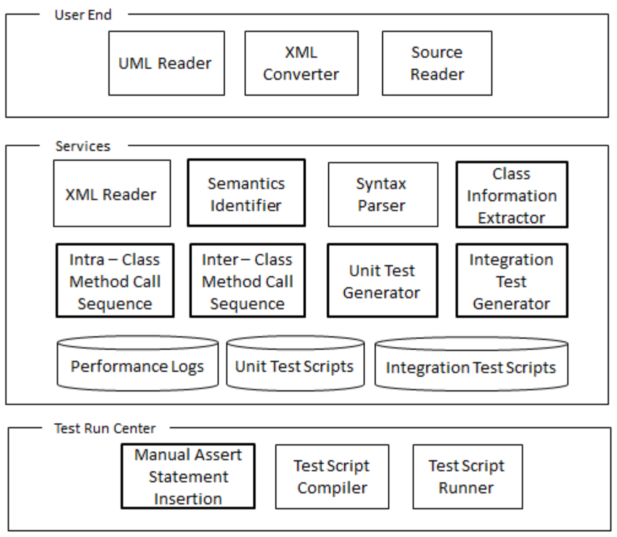


Fig. 2: The component stack of SSTF

The last layer is the Test Run Center. This layer is responsible for running the generated test cases.
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. 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 is introduced for this purpose. It produces XML files that can be later read by the framework. The third component of this layer is the Source Reader 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, Class Information Extractor, Intra-Class Method Call Sequence, Inter-Class Method Call Sequence, Unit Test Generator, and Integration Test Generator. Fig. 3 illustrates the interactions within those that follow after

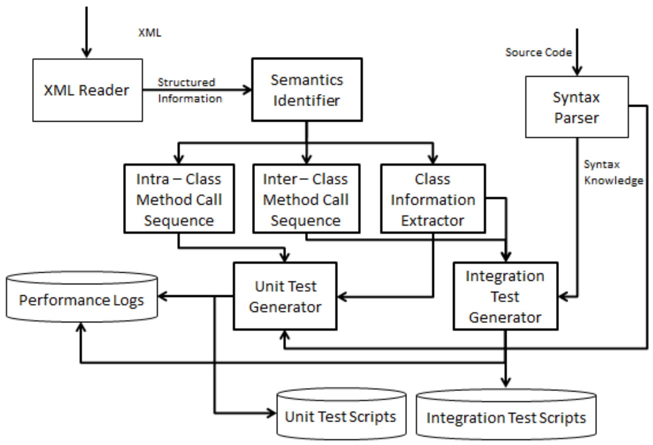
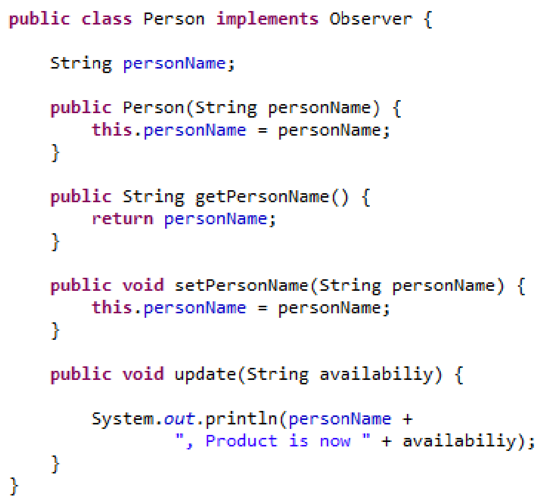


Fig. 3: The interaction among the components of the second layer

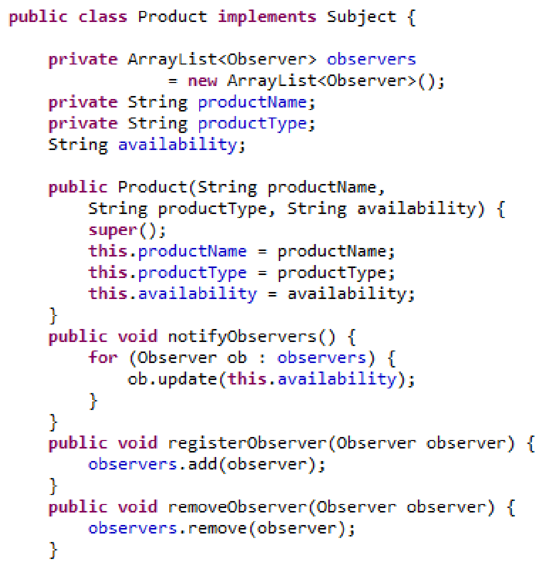
receiving information required from the first layer. The two supporting components of this layer are the XML Reader and Syntax Parser. The XML Reader receives files from the

XML Converter 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, Intra-Class Method Call Sequence 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 UML. The Intra-Class Method Call Sequence is in fact the state diagram information extractor. As an ideal state diagram wraps the state information of a specific class and traverse among the states in according to the method calls, the method call sequence inside the class can be identified effectively from state diagrams. This state information is stored by the Intra-Class Method Call Sequence and is later used for the generation of unit test cases. The next semantic supporter is the Inter-Class Method Call Sequence that works with the sequence diagrams of the software. Sequence diagram contains the class to class interaction through method calls. These interaction sequences are in fact the method call sequences among classes and are extracted by this component. These class interactions are later used to generate integration test cases.
The Unit Test Generator and Integration Test Generator are the most important components of this framework. These use the information produced so far by the other components and generate useful test cases. The Unit Test Generator takes input as follows: class information extracted from the class diagrams, internal class method calls from the state diagrams and syntax information from the source code. The class information defines the classes to be tested and also identifies mismatch between UML class information with classes extracted from source code for the inconsistency identification between UML and source. The state diagram method call information specifies the methods to be called and also the sequence of call. The syntax information provides the syntax of method call with parameters to be added in test script. These are afterward incorporated together to generate the unit test scripts; and is 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, this method call information works as input in the generation of integration tests. The process of incorporating this inter-class method call information with syntactic knowledge is similar to the Unit Test Generator. 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 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).



```
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);
    }
}
```

Fig. 4: Source code example (class: Person)

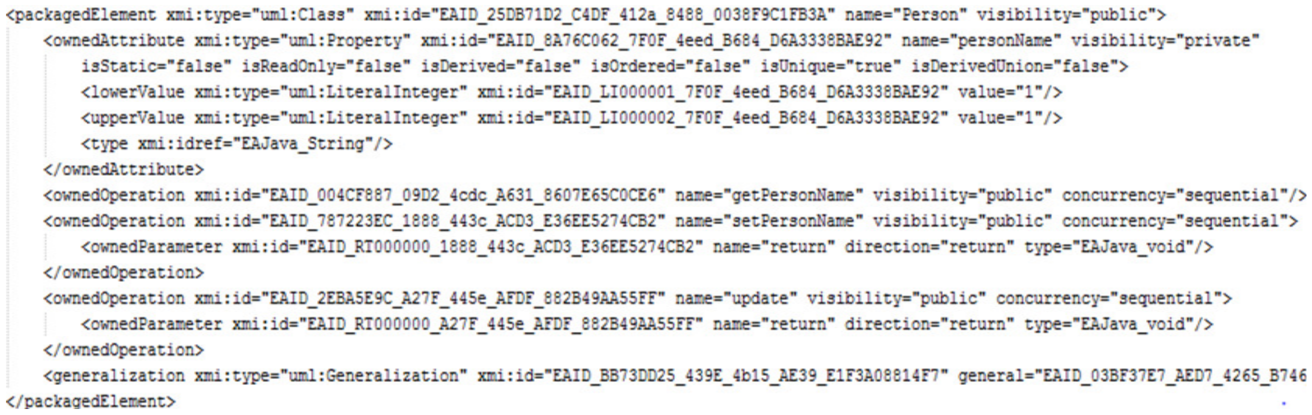


```
public class Product implements Subject {

    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);
    }
```

Fig. 5: Source code example (class: Product)

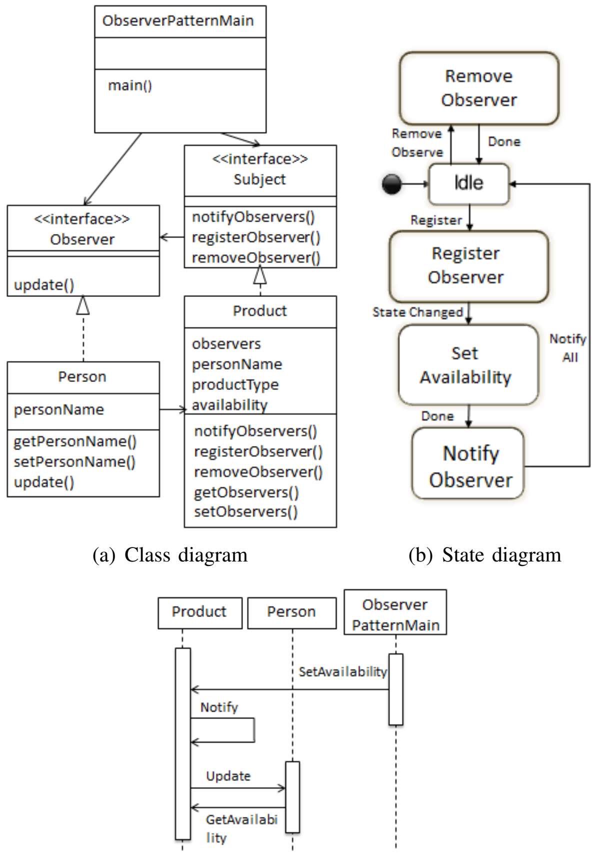
IV. Case Study

For an initial assessment of the competency, the SSTF was used on a simple java project illustrating observer pattern. The observer 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 [15]. The example project has a class called Person as observer class, a class called Product as subject class, and the Observer and Subject Interfaces. Fig. 4 and Fig. 5 show snapshots of the source code of the Person and Product class accordingly. The Source Reader takes this project source location as input and reads the java files to be processed by Syntax Parser in second layer. 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 UML diagrams are inputted in UML Reader next. These are converted to XML using the XML Converter. Enterprise Architect is used for this conversion purpose [16]. The class, state and sequence diagrams are shown in Fig. 6 and a portion of the produced XMLs are illustrated in Fig. 7, 8, 9.
The second layer takes the source and the produced XML



```
<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>
```

Fig. 7: Class XML (partial: Person class)

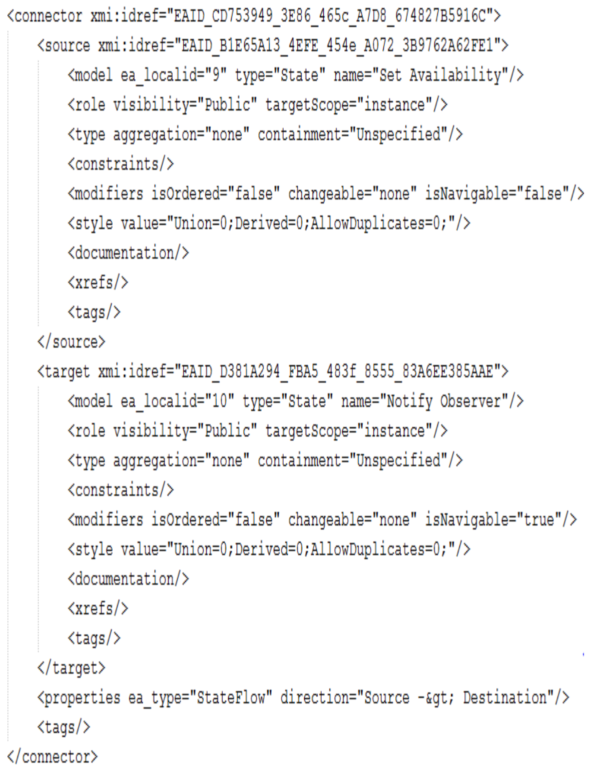


(a) Class diagram

(b) State diagram

(c) Sequence diagram

Fig. 6: The UML diagrams of the sample project



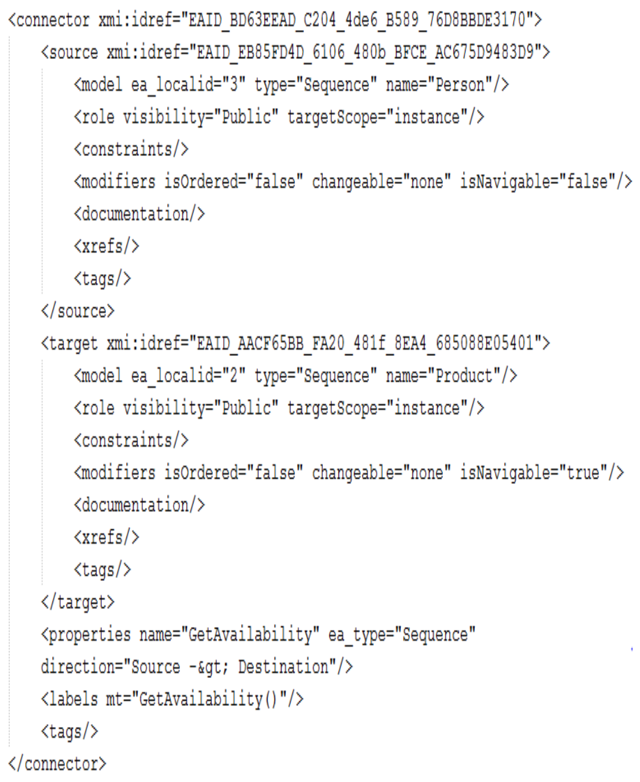
```
<connector xmi:idref="EAID_CD753949_3E86_465c_A7D8_674827B5916C">
    <source xmi:idref="EAID_B1E65A13_4EFE_454e_A072_3B9762A62FE1">
        <model ea_localid="9" type="State" name="Set Availability"/>
        <role visibility="Public" targetScope="instance"/>
        <type aggregation="none" containment="Unspecified"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
        <style value="Union=0;Derived=0;AllowDuplicates=0;"/>
        <documentation/>
        <xrefs/>
        <tags/>
    </source>
    <target xmi:idref="EAID_D381A294_FBA5_483f_8555_83A6EE385AAE">
        <model ea_localid="10" type="State" name="Notify Observer"/>
        <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;"/>
        <documentation/>
        <xrefs/>
        <tags/>
    </target>
    <properties ea_type="StateFlow" direction="Source -&gt; Destination"/>
    <tags/>
</connector>
```

Fig. 8: State XML (partial: connection of Set Availability with Notify Observer)

as input. The Class Information Extractor analyzes XML of the class diagram and identifies the class methods, variables as well as class association. The <owedAttriute> tags of the class XML refer class variables and the <owedOperation> tags refer class methods. The Intra- and Inter-Class Method Call Sequence extracts state and sequence diagram information similarly from their XML by parsing appropriate tags in it. At the same time the Syntax Parser parses the source for gathering the syntactic knowledge. These are then compared to check if there is any inconsistency between the syntax and semantic information. Then the syntax and semantic information is combined and the combined information is used by Unit and Integration Test Generator for the generation of unit and integration test accordingly. Sample unit and integration test suite snapshots are shown in Fig. 10 and Fig. 11.

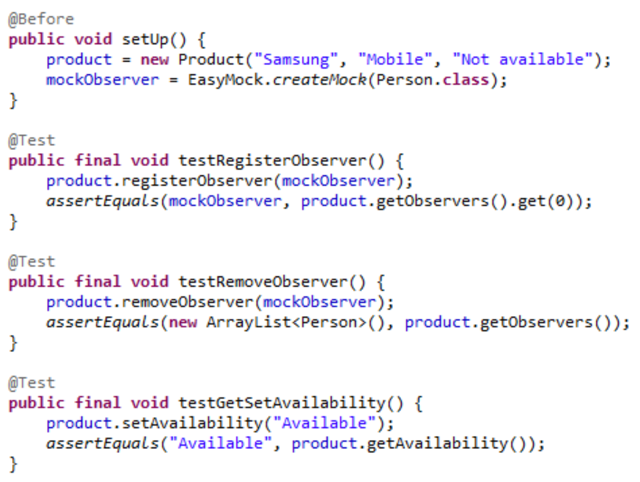
V. Conclusion

This paper 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. These 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



```
<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"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="false"/>
        <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"/>
        <constraints/>
        <modifiers isOrdered="false" changeable="none" isNavigable="true"/>
        <documentation/>
        <xrefs/>
        <tags/>
    </target>
    <properties name="GetAvailability" ea_type="Sequence"
    direction="Source -&gt; Destination"/>
    <labels mt="GetAvailability()"/>
    <tags/>
</connector>
```

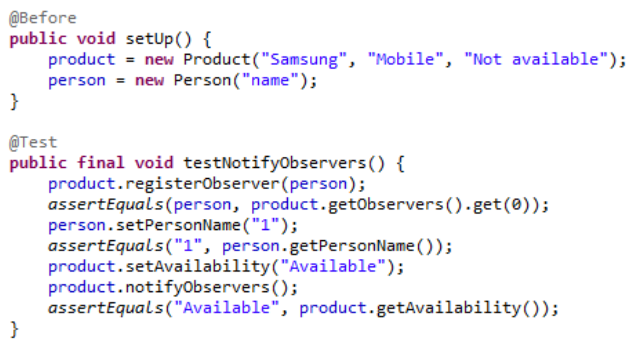
Fig. 9: Sequence XML (partial: GetAvailaility call)



```
@Before
public void setUp() {
    product = new Product("Samsung", "Mobile", "Not available");
    mockObserver = EasyMock.createMock(Person.class);
}

@Test
public final void testRegisterObserver() {
    product.registerObserver(mockObserver);
    assertEquals(mockObserver, product.getObservers().get(0));
}

@Test
public final void testRemoveObserver() {
    product.removeObserver(mockObserver);
    assertEquals(new ArrayList<Person>(), product.getObservers());
}

@Test
public final void testGetSetAvailability() {
    product.setAvailability("Available");
    assertEquals("Available", product.getAvailability());
}
```

Fig. 10: Unit test example

incorporation of semantics and syntax of software in test construction can improve the quality of generated test scripts. 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. This solution also has the ability to identify the effects of inconsistent UMLs and source while generating tests. A case study on a sample java project is shown here to assess the framework competence. The future challenge lies in assessing the performance of this framework in different platforms rather than Java only.



```
@Before
public void setUp() {
    product = new Product("Samsung", "Mobile", "Not available");
    person = new Person("name");
}

@Test
public final void testNotifyObservers() {
    product.registerObserver(person);
    assertEquals(person, product.getObservers().get(0));
    person.setPersonName("1");
    assertEquals("1", person.getPersonName());
    product.setAvailability("Available");
    product.notifyObservers();
    assertEquals("Available", product.getAvailability());
}
```

Fig. 11: Integration test example

References

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

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

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

[4] 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, May 2006, pp. 2–8.

[5] 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.

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

[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 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Mar. 2013, pp. 158–167.

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

[9] R. Heckel and M. Lohmann, "Towards model-driven testing," TACoS'03, International Workshop on Test and Analysis of Component-Based Systems, pp. 284–291, Apr. 2003.

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

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

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

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

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

[15] (2014, Jul.) Observer pattern - wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Observer_pattern

[16] (2014, Jun.) Enterprise architect - uml design tools and uml case tools for software development. [Online]. Available: http://www.sparxsystems.com/products/ea/