White-Box Testing Framework for Object-Oriented Programming based on Message Sequence Specification

Juan Ignacio Rodríguez Silva[1], Martín Larrea[1,2,3]
[1]Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur (DCIC-UNS)
[2]Instituto de Ciencias e Ingeniería de la Computación (UNS-CONICET)
[3]Laboratorio de I+D en Visualización y Computación Gráfica, (UNS-CIC Prov. de Buenos Aires)
nachorodriguez12@hotmail.com, mll@cs.uns.edu.ar

Abstract. Software is a crucial element in the functionality of devices and industry. Likewise, the operation of an enterprise or organization depends largely on the reliability of the software systems used for supporting the business process or particular tasks. The quality of software has become the most important factor in determining the success of products or enterprises. In order to accomplish a quality software product several methodologies, techniques, and frameworks have been developed, each of them tailored to specific areas or characteristics of the software under review. This paper presents a white-box testing framework for Object-Oriented Programming based on Message Sequence Specification. In the context of an object-oriented program, our framework can be used to test the correct order in which the methods of a class are invoked by its clients. The implementation of the framework is based on aspect-oriented programming.

Keywords: Software Verification Validation, White-Box Testing, Object-Oriented Programming, Message Sequence Specification, Aspect-Oriented Programming

1 Introduction

Verification and Validation (V&V) is the process of checking that a software system meets its specifications and fulfills its intended purpose. The software engineering community has acknowledged the importance of V&V process to ensure the quality of its software products. The V&V process, also known as software testing or just testing, is composed of V&V techniques. There are many different V&V techniques which are applicable at different stages of the development lifecycle. The two main categories of testing techniques are white-box and black-box. In the first one, the testing is driven by the knowledge and information provided by the implementation or source code. While in the second one the specification of the software, module, or function is used to test the object under review.

In 1994, Kirani and Tsai ([4]) presented a technique called Message Sequence Specification that, in the context of an object-oriented program, describes the correct order in which the methods of a class should be invoked by its clients. The method sequence specification associated with an object specifies all sequences of messages that the object can receive while still providing correct behavior. Daniels and Tsai [5] used the idea of Message Sequence Specification as a testing tool but without implementing a framework to support this technique.

We developed a framework for testing object-oriented programs based on Message Sequence Specification. Our framework can be used to test the correct order in which the methods of a class are invoked by its clients. The implementation of the framework was done using aspect-oriented programming, which allows to test the source code without the need to change any line of code in the program itself and even allows to run the tests while the user is using it.

In the remaining parts of this article, we first provide background V&V in general and them move on to review the concepts of Message Sequence Specification in the software development process. We continue with the presentation of aspect programming, a key part of our framework. Then, we describe our proposed framework. We later introduce an example of the use of such framework. We conclude with a brief discussion on limitations and advantages of our approach and the future work.

2 Background Review

2.1 Background on Verification & Validation

Software testing is involved in each stage of software life cycle, but how we test and what we test on each stage of software development is different, the nature and the goals of what is being tested are different. Based on [1], there are 8 types of testing in the life cycle: Unit testing is a code based testing which is performed by developers, this testing is mainly done to test each and individual units separately. The Unit testing can be done for small units of code, generally no larger than a class. Integration testing validates that two or more units work together properly, and inclines to focus on the interfaces specified in the low-level design. System testing reveals that the system works end-to-end in a production-like location to provide the business functions specified in the high-level design. Acceptance testing is conducted by business owners; the purpose of acceptance testing is to test whether the system does, in fact, meet their business requirements. Regression Testing is the testing of software after changes have been made to ensure that those changes did not introduce any new errors into the system. Alpha Testing is usually applied at the developer's site, with the presence of the developer. Beta Testing is done at the customer's site with no developer in site. Functional Testing is done for a finished application, its goal is to verify that the system provides all the required behavior.

In the context of V&V, black-box testing is often used for validation (i.e. are we building the right software?) and white-box testing is often used for

verification (i.e. are we building the software right?). In black-box testing, the test cases are based on the information from the specification. The software testers do not consider the internal source code of the test object. The focus of these tests is solely on the outputs generated in response to selected inputs and execution conditions. The software tester sees the software as a black box, where information is input to the box, and the box sends something back out. This can be done purely based on the requirement specification knowledge; the tester knows what to expect from the black box to send out and tests to make sure that the box sends out what it's supposed to send out.

Oppositely, white-box testing, also called structural testing, designs test cases based on the information derived from the source code. White-box testing is concern with the internal mechanism of a system, it mainly focuses on the control flow or data flow of the program. White-box and black-box testing are considered to complement each other. In order to test software correctly, it is essential to generate test cases from the specification and source code. This means that we must use white-box and black-box techniques on the software under development.

Each test technique, both white-box and black-box, must describe a test model and, at least, one coverage criteria. Test models describe how to generate test cases, it can be a graph, a table or a set of numbers. Coverage criteria are used to steer and stop the test generation process ([2]), they are usually boolean conditions. They have widely accepted means of assessing the quality of a test ([3]).

The same testing technique that we classified as white or black box, can be arranged as static or dynamic techniques. Static testing are those techniques where the code is not executed, it can be analyzed manually or by a set of tools. This type of testing checks the code, requirement documents, and design documents. Dynamic testing is done when the code is executed. Dynamic testing is performed when the code being executed is input with a value, the result or the output of the code is checked and compared with the expected output.

2.2 Background on Message Sequence Specification

In 1994, Kirani and Tsai ([4]) presented a technique called Message Sequence Specification that, in the context of an object-oriented program, describes the correct order in which the methods of a class should be invoked by its clients. The method sequence specification associated with an object specifies all sequences of messages that the object can receive while still providing correct behavior.

Their strategy used regular expressions to model the constraints over the correct order of the invocation of the methods i.e. the regular expression is the test model. Method names were used as the alphabet of a grammar which was then used to statically verify the program's implementation for improper method sequences. A runtime verification system identifies incorrect method invocations by checking for sequence consistency with respect to the sequencing constraints.

If a class C has a method M_1, this is noted as C_{M_1}. Sequence relationships between two methods were classified into three categories, sequential, optional,

and repeated. If the method M_1 of C should be invoked before the method M_2 of the same class, then this relationship is sequential and is represented as

$$C_{M_1} \bullet C_{M_2}$$

If one, and only one of the methods M_1 and M_2 can be invoked then this relationship is optional and is represented as

$$C_{M_1} | C_{M_2}$$

Finally, if the method M_1 can be invoked many times in a row then this is a repeated relationship and is represented as

$$(C_{M_1})^*$$

For example, if a class X has three methods *create*, *process* and *close*, a possible sequencing constraint based on Message Sequence Specification could look like

$$X_{create} \bullet (X_{process})^* \bullet X_{close}$$

If class X is part of a larger system S, then we could statically check the source code of S to see if all calls to X's methods follow the defined grammar. If a static analysis is not enough, we could implement a runtime verification system that tracks all calls to X's methods and checks dynamically the sequence of calls against the grammar.

This technique can also be used to test the robustness of a system. Continuing with the class X as an example, we can use the defined grammar to create method sequences that are not a derivation from the grammar, i.e. incorrect sequences methods. These new sequences can be used to test how the class handles a misuse. For example, how does the class X respond to the following sequence of calls:

$$X_{create} \bullet X_{close} \bullet X_{process}$$

Testing with the sequences generated by the grammar and with those that were not are two variations presented in [5], a work that extended the research done in [4].

2.3 Background in Aspect Oriented Programming

Aspect Oriented Programming [6] (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with *set*". This allows behaviors that are not central to the business logic (such as logging or testing) to be added to a program without cluttering the code, core to the functionality.

AOP entails breaking down program logic into distinct parts (so-called concerns). Nearly all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g., functions, procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. Some concerns cut across multiple abstractions in a program, and defy these forms of implementation. These concerns are called cross-cutting concerns or horizontal

concerns. Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every logged part of the system. Logging thereby crosscuts all logged classes and methods.

See for example Listing 1.1, a simple Java class for a bank account. If we need to log all the events in the account one way to do it is as the listing shows. The main disadvantages of this approach is that we are mixing the logic of the bank account class with the requirement of logging its events. By using AspectJ we can create an aspect, as in Listing 1.3 while the bank account class remains simpler 1.2. With this two classes, every time there is a call to *deposit* or *withdraw* the jre will execute the methods in the AspectJ aspect.

Listing 1.1. Classic example

```
public class Account {
protected int amount;

public Account()
{ this.amount = 0; }

public void deposit( int _amount
{ this.amount += _amount;
 Log.put("deposit for" + _amount);
 }

public void whitdraw( int _amount )
{ this.amount -= _amount;
 Log.put("whitdraw for" + _amount);
 } }
```

Listing 1.2. Clear code

```
public class Account {
protected int amount;

public Account()
{ this.amount = 0; }

public void deposit( int _amount
{ this.amount += _amount;
 }

public void whitdraw( int _amount )
{ this.amount -= _amount;
 } }
```

Listing 1.3. AspectJ code

```
public aspect AspectLogic {

before(int _amount):
call(void Account.deposit(int))
 && args(_amount) {
Log.put("deposit for " + _amount);
}

before(int _amount):
call(void Account.whitdraw(int))
 && args(_amount) {
Log.put("whitdraw for " + _amount);
 } }
```

By using AspectJ we can add checks to a source code with out the need to change the code itself. This is a very appealing feature in the context of software verification and validation. For more on AOP we recommend the book [6].

3 Previous Work on Testing with Message Sequence Specification

The technique presented in 1994 by Kirani and Tsai ([4]) was follow in 1999 by Daniels and Tsai in [5]. Also in 1999, Tsai et al. presented [8] where Message Sequence Specification was used to create template scenarios than later were used to create test cases. In 2003, Tsai used the technique [7] as a verification mechanism to the UDDI servers in the context of Web Services. In 2014, [9] introduced a Java-based tool for monitoring sequences of method calls, similar to our goal in this work they used annotations instead of AOP.

4 Our Proposal

Our goal in this work is to present a testing framework for object-oriented source code based on Message Sequence Specification by using AOP. AOP will allow us to create our test cases without the need to modify the source code, and those test cases will run automatically with each run of the program under test. The use of message sequence specification will allow the developer of each class to describe a grammar that will represent the correct behavior of such class. The framework will take these grammars, run the program and check that the methods are used according to the developer specification. We wanted to provide an easy to use framework, with an easy to read and understand representation of the methods correct usage. More particularly, we wanted a framework that the developer can use, without the need for a testing specialist.

The first thing the developer must do to use the framework is to create the regular expression associated with its class. This is the correct behavior or order in which the methods of the class should be called. In order to express this

grammar in a simple way, the developer is encouraged to use simple symbols that represent the methods. This means that it is not required to use the actual names of the methods in the grammar. But, in order to keep the grammar easy to read, at some point the developer must create a map between actual methods name and their short version.

Lets use Listing 1.4 as an example. Again we have a Bank Account class, but a bit more complex. In this case, the correct order to use the Account is: first the account must be created and then it must be verified. The first money movement in the account must be a deposit, after that we can deposit or withdraw money. Finally, the account must be closed.

Listing 1.4. Classic example

```
public class Account {
protected int amount;
protected boolean verify;

public Account()
{ this.amount = 0;
this.verify = false; }

public void verify()
{ this.verify = true; }

public void deposit( int _amount )
{ if (this.isVerify()) this.amount += _amount; }

public void whitdraw( int _amount )
{ if (this.isVerify()) this.amount -= _amount; }

public void close()
{ this.amount = 0;
this.verify = false; }

public boolean isVerify()
{ return this.verify; } }
```

Based on Message Sequence Specification, the grammar for the correct use of this class is as follows:

$$create \bullet verify \bullet deposit \bullet (deposit|withdraw)^* \bullet close$$

or as a more simpler expression, as:

$$c \bullet v \bullet d \bullet (d|w)^* \bullet x$$

As shown the names that we used in the grammar not necessarily have to be the actual methods name. This can be used to enhance the readability of the grammar, but in order to use this as it is, we need to map those name to the actual methods. For this goal, the framework offers a $Map < String, String >$

map; variable where the mapping is store. Before we can tell the framework about our grammar for the Bank Account class, we must first input the mapping as shown in the Listing 1.5. We then can introduce the grammar to the framework (Listing 1.6). With this two steps completed, it is now possible to execute the main program and let the framework check, at runtime, in which order are the methods of the Bank Account class being called.

Listing 1.5. Classic example

```
map.put("bank.Account.<init>", "c");
map.put("bank.Account.verify", "v");
map.put("bank.Account.deposit", "d");
map.put("bank.Account.withdraw", "w");
map.put("bank.Account.close", "x");
```

Listing 1.6. Classic example

```
Pattern regex = Pattern.compile("cvd(d|w)*x");
Matcher matcher = regex.matcher("");
```

Once the main program is running, our framework will keep a log of how the methods in the grammar are called. If a method call doesn't follow the grammar form, the framework will issue an alert in the console output. A trace of all the call will be output to the console as well.

The usefulness of this framework can be emphasized in the following section, where we use our development to find an error in a real case, an application developed in our laboratory.

5 Case Study. Rock.AR, a software solution for point counting

Point counting is the standard method to establish the modal proportion of minerals in coarse-grained igneous, metamorphic, and sedimentary rock samples. This requires observations to be made at regular positions on the sample, namely grid intersections. At each position, the domain expert decides to which mineral the respective grid point and its local neighborhood belongs. By counting the number of points found for each mineral, it is possible to calculate the percentages that these values represent of the total counted points. These percentages represent the approximate relative proportions of the minerals in a rock, which is a 2D section of a 3D sample. Rock.AR ([11]) is a visualization tool with a user-friendly interface that provides a semiautomatic point-counting method. It increases the efficiency of the point-counting task by reducing the user cognitive workload. This tool automates the creation of the grid used to define the point positions. This grid is overlaid on a predetermined image of a sample, allowing the count of minerals identified at the intersections of the grid lines. This method significantly reduces the time required to conduct point counting, it does not require an expensive ad hoc device to perform the job, and

it improves the consistency of counts. Among the methods available in the main class of this application, there are three of interest for us. Those are LoadSample(), MoveSelectedCell() and AddNewRockType(). In this program, the correct used of these methods is: First it must be at least one call to LoadSample(), then before a new rock type can be added, a point (also known as a cell) must be selected. We can create a grammar from this as such:

$$(LoadSample \bullet LoadSample^* \bullet (MoveSelectedCell \bullet MoveSelectedCell^* \bullet AddNewRockType^*)^*)^*$$

or in a more simpler way:

$$(l \bullet l^* \bullet (m \bullet m^* \bullet a^*)^*)^*$$

A particular observation of this grammar is that, as defined by the developer, before adding a new mineral type at least one cell must be selected, and before that a sample must be load. Using our framework, we created this grammar and we enter it into the framework. After this we ran the program several times and used Rock.AR for a while. In the third run of the program, the test framework detect an error and output the sequence of calls that didn't follow the grammar. The sequence was: :

$$l \bullet m \bullet m \bullet a \bullet l \bullet l \bullet a$$

The last three call on the sequence are the interesting ones. According to these, it was possible to call the method AddNewRockType() without having to call MoveSelectedCell() first. Using this information we discover that when the user load a second sample in the programm, there were two variables that were not initialized. Because of this, it was possible to call AddNewRockType() right after LoadSample() something that was not allow in the grammar. Before the use of our testing framework there was no evidence of the error found. The error does not generate any exception in execution, this is because since the variables were already initialized their old values were taken and the program continued its execution.

6 Conclusions & Future Work

In a world where technology is part of every person's daily life, software is a crucial element. The quality of software has become the most important factor and the developer need tools to assist them in their work in order to achieve such high quality. In this work, we presented a framework for white-box testing. Our framework combines Message Sequence Specification with Aspect Oriented Programming in order to achieve a tool to test the correct order in which methods in a class are being called. As we stated earlier, our goal was to create an easy to use framework for testing. A framework that the developer could use on its own code and a framework that will not interfere with the code under review. The intention of this framework is to find a particular type of error, which occurs when the order in which the methods of a class are called is relevant. As it was demonstrated in the case of study, the framework allows to detect errors that otherwise would be difficult to determine.

As for future work, a more expressive framework is under consideration. For the moment, we can only describe the order in which the methods are being called but not any information on how are they call or the state of the class instance. It would be useful to say that a method x must be called after method y if the value of an attribute is equal to 0.

Acknowledgment

This work was partially supported by the following research projects: PGI 24/N037 and PGI 24/ZN29 from the Secretaría General de Ciencia y Tecnología, Universidad Nacional del Sur, Argentina.

References

1. Jorgensen, Paul C. Software testing: a craftsman's approach. CRC Press, 2013.
2. Weileder, Stephan. Test models and coverage criteria for automatic model-based test generation with UML state machines. Diss. Humboldt University of Berlin, 2010.
3. Friske, Mario, Bernd-Holger Schlingloff, and Stephan Weileder. "Composition of Model-based Test Coverage Criteria." MBEES. 2008.
4. Kirani, Shekhar H., and W. T. Tsai. Specification and verification of object-oriented programs. Diss. University of Minnesota, 1994.
5. Daniels, F. J., and K-C. Tai. "Measuring the effectiveness of method test sequences derived from sequencing constraints." Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings. IEEE, 1999.
6. Laddad, Ramnivas. AspectJ in action: practical aspect-oriented programming. Dreamtech Press, 2003.
7. Tsai, W. T., Paul, R., Cao, Z., Yu, L., Saimi, A. (2003, January). Verification of web services using an enhanced UDDI server. In Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on (pp. 131-138). IEEE.
8. Tsai, W. T., Tu, Y., Shao, W., Ebner, E. (1999). Testing extensible design patterns in object-oriented frameworks through scenario templates. In Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International (pp. 166-171). IEEE.
9. Nobakht, B., de Boer, F. S., Bonsangue, M. M., de Gouw, S., Jaghoori, M. M. (2014). Monitoring method call sequences using annotations. Science of Computer Programming, 94, 362-378.
10. Bai, X., Lu, H., Zhang, Y., Zhang, R., Hu, L., Ye, H. (2011, July). Interface-Based automated testing for open software architecture. In Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual (pp. 149-154). IEEE.
11. Larrea, M. L., Castro, S. M., Bjerg, E. A. (2014). A software solution for point counting. Petrographic thin section analysis as a case study. Arabian Journal of Geosciences, 7(8), 2981-2989.