White-Box Testing Framework for Object-Oriented Programming. An approach based on Message Sequence Specification and Aspect Oriented Programming

Martín L. Larrea[†], Juan I. Rodríguez Silva, Matías N. Selzer, and Dana K. Urribarri

Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur (DCIC-UNS)
Instituto de Ciencias e Ingeniería de la Computación (UNS-CONICET)
Laboratorio de I+D en Visualización y Computación Gráfica, (UNS-CIC Prov. de Buenos Aires)
mll@cs.uns.edu.ar, nachorodriguez12@hotmail.com,
matias.selzer@cs.uns.edu.ar, dku@cs.uns.edu.ar

Abstract. The quality of software has become one of 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 one developed and 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 and Aspect Oriented Programming. 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.

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 the V&V process to ensure the quality of its software products. The V&V process, also known as software testing or just testing, consist 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

[†]Corresponding author

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 [1] presented a technique called Message Sequence Specification (MSS) that, in the context of an object-oriented program, describes the correct order in which the methods of a class should be invoked. 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 [2] used the idea of MSS as a testing tool but without implementing a framework to support this technique.

In an earlier publication, we published a framework for testing object-oriented programs based on MSS [3]. Our framework could be used to test the correct order in which the methods of a class are invoked. The framework was implemented using Aspect-Oriented Programming (AOP). AOP provides transparent testing, i.e., the source code can be tested without additional modifications and while the software is running. Since this first version of the framework was limited to test only one instance of a class, it was suitable only for usages where the singleton pattern was appropriate. It could also be used to test the behavior of GUI elements. In this paper, we present a new version of the framework, one that can test multiple instances of a class using MSS. We also update the previous work with new references and expand our case study section.

The rest of the paper is structured as follows. Section 2.1 provides background information about V&V, concepts of MSS in the software development process, and AOP, which constitutes our framework's core. The framework's first implementation [3] is presented next, followed by the new proposed framework and how it differs from the one presented previously. We later introduce two examples of the framework's use. Finally, we conclude with a brief discussion on the limitations and advantages of our approach and the future work.

2 Background

2.1 Verification & Validation

Software testing is involved in every stage of software life cycle, however, how the test is performed and what is tested at each stage is different, as well as the nature and goals of what is being tested. Jorgensen [4] describes 8 types of testing in the life cycle: Unit testing is a code-based testing performed by developers to test each individual unit separately. The Unit testing can be used for small units of code, generally no larger than a class. Integration testing validates that two or more units work together properly, and focuses 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 complies with the 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. Functional Testing is done for a finished application 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, who knows in advance the expected outcome of the black box, tests the software to guarantee that the actual result complies with the expected one.

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

Both white-box and black-box test techniques must describe a test model and at least one coverage criteria. Test models describe how to generate test cases, and can be a graph, a table or a set of numbers. Coverage criteria, on the other hand, are usually boolean conditions to steer and stop the test generation process [5]. In the literature, coverage criteria are widely accepted for assessing the quality of a test [6].

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 in which the code is not executed. In this case, the code 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, and the result or the output of the code is checked and compared with the expected output.

2.2 Message Sequence Specification

In 1994, Kirani and Tsai [1] presented a technique called Message Sequence Specification (MSS) 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 the expression 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.

According to Kirani's specification, 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

$$C_{M_1} \bullet C_{M_2} \tag{1}$$

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

$$C_{M_1} | C_{M_2} \tag{2}$$

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

$$(C_{M_1})^* \tag{3}$$

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

$$X_{create} \bullet (X_{process})^* \bullet X_{close} \tag{4}$$

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 expression. 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 its regular expression.

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 regular expression to create method sequences that are not a derivation from it, 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?:

$$X_{create} \bullet X_{close} \mathrm{X}_{process} \tag{5}$$

Daniels & Tsai [2] extended the work of Kirani et al. [1] by testing with sequences that were both generated by the expression and not. Also in 1999, Tsai et al. [7] presented a work where MSS was used to create template scenarios than later were used to create test cases. In 2003, Tsai [8] used MSS as a verification mechanism to the UDDI servers in the context of Web Services. In 2014, a Java-based tool for monitoring sequences of method calls was introduced [9], it

had similar objectives as our work but they used annotations instead of AOP. We introduced MSS as an approach for testing visualizations interactions [10] in 2018; and finally, Turner [11] used sequence specification for testing GUI in 2019.

2.3 Aspect Oriented Programming

Aspect Oriented Programming [12] (AOP) is a programming paradigm designed to increase modularity, based on the separation of cross-cutting concerns. With AOP "pointcut" specifications, the already implemented software can incorporate additional behavior without changing the source code. A pointcut declares a specification, for instance, "log all function calls when the function's name begins with *set*". Hence, behaviors that are not central to the business logic (such as logging or testing) can be added to a program without cluttering the code.

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 is an example of a crosscutting concern because a logging strategy necessarily affects every logged part of the system. Logging thereby cross-cuts 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 are that we are mixing the logic of the bank account class with the requirement of logging its events. By using AspectJ, an implementation of AOP for Java, we can create an aspect, as in Listing 1.3 while the bank account class remains simpler 1.2. With these 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 withdraw( int _amount ) {
```

```
        this.amount -= _amount;
        Log.put("withdraw 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 withdraw( 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.withdraw(int)) && args(_amount) {
            Log.put("withdraw for " + _amount);
        }
}
```

By using AspectJ we can add new behavior to a source code without the need to change the code itself. This is a very appealing feature in the context of software verification and validation. More information about AOP can be found in [12].

3 Our Testing Framework

Our goal in this work is to present a testing framework for object-oriented source code based on MSS using AOP. AOP allows us to create test cases without the need to modify the source code, and those test cases can run automatically with each run of the program under test. The use of MSS allows the developer of a class to describe a a regular expression that represents the correct behavior of such class. The framework takes each of these expressions, runs the program and

checks 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 for the correct usage of the methods. Particularly, the framework was designed to be used by any developer, without needing a testing specialist.

The first thing the developer must do to use the framework is to create the regular expression associated with the class under test. This regular expression must specify the correct behavior or order in which the methods of the class should be called. In order to express this in a simple way, the developer must use simple symbols (i.e. characters) to represent each method. This means that the actual names of the methods are not used in the expression. But, to be able to interpret it at some point the developer must create a map between the actual methods' names and their corresponding symbol. The regular expression and the map between methods and symbols are set in the *TestingSetup.java* class. The framework consists of two main components, an aspect, and a java class. The aspect is named *TestingCore.aj* and it contains the implementation of the framework's logic.

Listing 1.4 shows an example with a more complex Account class. 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. Once the account is closed, no more operations are allowed.

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 withdraw( 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 MSS, the regular expression for the correct use of this class is as follows:

$$create \bullet verify \bullet deposit \bullet (deposit|withdraw)^* \bullet close \tag{6}$$

or as a simpler expression:

$$c \bullet v \bullet d \bullet (d|w)^* \bullet x. \tag{7}$$

Instead of using the actual methods' names, a set of corresponding symbols are used to enhance the readability of the regular expression. For this reason, the framework allows the developer to map such methods' names to character symbols. After that, the developer is able to input the regular expression for the class being tested. Listing 1.5 shows the configuration of the *TestingSetup.java* class for the actual example. With these two steps completed, the framework checks at runtime that the methods of the Account class are being called in the correct order.

Listing 1.5. Class TestingSetup

```
//Specification of the test class
TestingCore.targetClass = Account.class.toString();

//Definition of the methods and their
// corresponding symbols
TestingCore.mapObjectsToCallSequence = new HashMap<>();
TestingCore.mapMethodsToSymbols =
    new HashMap<String, String>();
TestingCore.mapMethodsToSymbols.
    put("main.Account.<init>", "c");
TestingCore.mapMethodsToSymbols.
    put("main.Account.verify", "v");
TestingCore.mapMethodsToSymbols.
    put("main.Account.deposit", "d");
TestingCore.mapMethodsToSymbols.
    put("main.Account.withdraw", "w");
TestingCore.mapMethodsToSymbols.
    put("main.Account.close", "x");

//Definition of the regular expression
TestingCore.regularExpression =
    Pattern.compile("cvd(d|w)*x");

//Initializing the regular expressions controller
TestingCore.matcher =
    TestingCore.regularExpression.matcher("");
```

If a method call does not match the regular expression, the framework aborts the execution of the program. The hashcode of the object causing the error, the regular expression, the sequence of called methods, and the name of last called method are issued via the standard output for the developer to understand the error.

Listing 1.6 shows an invalid use of the methods in the Account class; particularly the *a1.verify()* line at the end of the program does not comply with the regular expression provided for the Account class. In this case, the framework issues the message shown in listing 1.7 and aborts the program.

Listing 1.6. Invalid used of class Account

```
public static void main(String[] args) {
        Account a1 = new Account();
        a1.verify();
        a1.deposit(1000);
        a1.deposit(4000);
        a1.withdraw(3000);
        a1.verify();
        a1.close(); }
```

Listing 1.7. Error message by the framework

```
------------------------------
---       ERROR FOUND       ---
------------------------------
Object Code: 507084503
Method Executed: main.Account.verify
Regular Expression: cvd(d|w)*x
Execution Sequence: cvddwv
------------------------------
-----  SYSTEM ABORTING... -----
------------------------------
```

The next section shows the framework usefulness in a real-life situation. By using this framework, we found two errors in an application developed in our research group.

4 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 to make observations to be made at regular positions on the sample, namely grid intersections. For each position, the domain expert decides which mineral corresponds to the respective grid point and its local neighborhood. By counting the number of points found for each mineral, it is possible to calculate

the percentage that each value 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 [13] 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. The grid is overlaid on a sample image and allows to identify and count minerals 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.

4.1 First Detected Bug

The main class of this application contains three important methods:

- LoadSample() loads the rock thin section sample.
- AddNewRockType() links a type of mineral with the selected cell in the grid.
- MoveSelectedCell() selects a cell in the grid.

First, at least one sample must be loaded. Then, before linking a mineral type to a cell, a point (also known as a cell) must be selected.

The following regular expression can be defined:

$$(LoadSample \bullet LoadSample^* \bullet (MoveSelectedCell \bullet MoveSelectedCell^* \bullet AddNewRockType^*)^*)^* \quad (8)$$

or in a simpler way:

$$(l \bullet l^* \bullet (m \bullet m^* \bullet a^*)^*)^* \quad (9)$$

Listing 1.8. Class TestingSetup for Rock.AR

```
//Specification of the test class
TestingCore.targetClass = ViMuGenMain2.class.toString();

//Definition of the methods and their
// corresponding symbols
TestingCore.mapObjectsToCallSequence = new HashMap<>();
TestingCore.mapMethodsToSymbols =
    new HashMap<String, String>();
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.main.
ViMuGenMain2.LoadSample", "l");
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.main.
ViMuGenMain2.MoveSelectedCell", "m");
TestingCore.mapMethodsToSymbols.
```

```
    put("ar.edu.uns.cs.vyglab.vimuge.main.
ViMuGenMain2.AddNewRockType", "a");

//Definition of the regular expression
TestingCore.regularExpression =
    Pattern.compile("ll*(mm*a*)*)*");

//Initializing the regular expressions controller
TestingCore.matcher =
    TestingCore.regularExpression.matcher("");
```

A particularity of this regular expression is that, between loading a sample and adding a new rock type, at least one cell must be selected. We created this regular expression and input it into the framework as shown in Listing 1.8. After defining and providing this expression to the framework we used Rock.AR several times. After running the program several times, the test framework detected an error and output the sequence of calls that did not comply with the regular expression. The sequence was:

$$l \bullet m \bullet m \bullet a \bullet l \bullet l \bullet a \tag{10}$$

The last three symbols in the sequence indicated that the application allowed calling AddNewRockType() right after LoadSample(), i.e. without calling MoveSelectedCell() in between. This calling sequence was not allowed in the regular expression. With this information we discovered that, when the user loaded a second sample, two variables were not re-initialized, causing this incorrect behavior. Before using our framework, there was no evidence of this error. The application did not generate any exception since the program was using old values for those not re-initialized variables.

4.2 Second Detected Bug

In order to help the domain expert to visualize the grid intersection, a colored grid is drawn on top of the mineral image. Each intersection is represented in the implementation by an instance of the class *SampleCell*. Usually, a mineral image requires around 1250 cells. The *SampleCell* class offers methods to set and get the dimensions of the cell and the mineral type. However, once the dimensions are set, they can not be changed. This is because the size of each cell is calculated by a math equation and it can not be set by the user. Since the mineral type of each cell depends on the user's subjective appreciation, it can be defined more than once. This behavior is defined in the following regular expression:

$$\begin{aligned} &CreateSampleCell \bullet (setDimension \bullet setRockType| \\ &\quad setRockType \bullet setDimension) \bullet (setRockType|getRockType| \\ &\qquad\qquad getDimension)^* \end{aligned} \tag{11}$$

or in a simpler way:

$$x \bullet (d \bullet r | r \bullet d) \bullet (r|w|e)^* \tag{12}$$

As described in this expression, after the creation of the sample cell, both the dimension and mineral type must be set but in no particular order. After that, the mineral type can be set and get as many times as necessary. The dimension, however, can only be retrieve. Listing 1.9 shows how the class *TestingSetup* was modified.

Listing 1.9. Class TestingSetup for Rock.AR

```
//Specification of the test class
TestingCore.targetClass = SampleCell.class.toString();

//Definition of the methods and their
// corresponding symbols
TestingCore.mapObjectsToCallSequence = new HashMap<>();
TestingCore.mapMethodsToSymbols =
    new HashMap<String, String>();
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.<init>", "x");
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.SetDimension", "d");
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.GetDimension", "e");
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.SetRockType", "r");
TestingCore.mapMethodsToSymbols.
    put("ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.GetRockType", "w");

//Definition of the regular expression
TestingCore.regularExpression =
    Pattern.compile("x(dr|rd)(wer)*");

//Initializing the regular expressions controller
TestingCore.matcher =
    TestingCore.regularExpression.matcher("");
```

As in the previous study case, we ran Rock.AR several times to test the last regular expression. The framework detected an error and output the message shown in Listing 1.10. By inspecting the execution sequence we could see that a mineral type was retrieved from a sample cell without being previously set. This error occurred every time the user loaded a previously saved work.

When users saved their work, Rock.AR saved to a file only the sample cells with a mineral type assigned. All sample cell (with or without a mineral type assigned) are stored in an array, and the system keeps a second array to index those sample cell with an assigned mineral. When the work is saved to a file, the application goes through this last array and retrieves those cells with an assigned mineral. The method responsible for this operation had an error in a boolean condition: instead of iterate until $amount-1$, it went until $amount$. This did not produce a runtime error because we keep the array as large as possible so, under certain conditions, $array[amount]$ was an actual sample cell. The problem was that this cell did not have an assigned mineral, so a value was obtained without first assigning it.

Listing 1.10. Error message by the framework on Rock.AR

```
------------------------------
---        ERROR FOUND       ---
------------------------------
Object Code: 1061804750
Method Executed: ar.edu.uns.cs.vyglab.vimuge.data.
SampleCell.GetRockType
Regular Expression: x(dr|rd)(wer)*
Execution Sequence: xdw
------------------------------
-----  SYSTEM ABORTING... -----
------------------------------
```

The framework is available for downloading[1]. The source code is available and licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

5 Conclusions & Future Work

In a world where technology is part of everyone's life, software is a crucial element. The quality of software has become one of the most important factors and developers need tools to assist them in their work in order to achieve such high quality. In this work, we present a framework for white-box testing. Our framework combines MSS with Aspect Oriented Programming in order to create a tool that tests 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 the developers to test source code without any modification of the code under review. The goal of this framework is to find those vulnerabilities that allow the expected execution order of the methods of a class to be broken. As it was shown in the case studies, the framework helps to detect errors that otherwise would be difficult to find. Future work will consider a more expressive framework. For the moment, the framework can only test the order in which methods of a class

[1] http://cs.uns.edu.ar/~mll/lapaz/

are being called. However, the actual values of the parameters of a method or the inner state of the instance can also be relevant in the execution order. For example, a class could require a method x to be called after method y if the value of a particular attribute is equal to 0.

At the moment, the framework can only test multiple instances of the same class. We will consider extending the framework to test multiple classes at the same time and the order between methods of different classes.

Finally, whenever an error is found, the framework aborts the execution of the program. A future improvement will allow the developer to specify for each particular error if the execution should continue or be aborted.

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. Kirani, S., Tsai, W.T.: Specification and verification of object-oriented programs. Tech. rep., Computer Science Department, University of Minnesota (1994)
2. Daniels, F., Tai, K.: Measuring the effectiveness of method test sequences derived from sequencing constraints. In: Proceedings of Technology of Object-Oriented Languages and Systems-TOOLS 30 (Cat. No. PR00278), pp. 74–83. IEEE (1999). doi:10.1109/TOOLS.1999.787537
3. Silva, J.I.R., Larrea, M.: White-box testing framework for object-oriented programming based on message sequence specification. In: XXIV Congreso Argentino de Ciencias de la Computación (Tandil, 2018)., pp. 532–541 (2018)
4. Jorgensen, P.C.: Software testing: a craftsman's approach. Auerbach Publications (2013). doi:10.1201/9781439889503
5. Weißleder, S.: Test models and coverage criteria for automatic model-based test generation with uml state machines. Ph.D. thesis, Humboldt University of Berlin (2010)
6. Friske, M., Schlingloff, B.H., Weißleder, S.: Composition of model-based test coverage criteria. In: MBEES, pp. 87–94 (2008). doi:10.1.1.459.6744
7. Tsai, W.T., Tu, Y., Shao, W., Ebner, E.: Testing extensible design patterns in object-oriented frameworks through scenario templates. In: Proceedings. Twenty-Third Annual International Computer Software and Applications Conference (Cat. No. 99CB37032), pp. 166–171. IEEE (1999). doi:10.1109/CMPSAC.1999.812695
8. Tsai, W.T., Paul, R., Cao, Z., Yu, L., Saimi, A.: Verification of web services using an enhanced uddi server. In: Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003)., pp. 131–138. IEEE (2003). doi:10.1109/WORDS.2003.1218075
9. Nobakht, B., de Boer, F.S., Bonsangue, M.M., de Gouw, S., Jaghoori, M.M.: Monitoring method call sequences using annotations. Science of Computer Programming **94**, 362–378 (2014). doi:10.1016/j.scico.2013.11.030

10. Larrea, M.L.: Black-box testing technique for information visualization. sequencing constraints with low-level interactions. Journal of Computer Science & Technology **17** (2017)
11. Turner, J.D.: Supporting interactive system testing with interaction sequences. Ph.D. thesis, The University of Waikato (2019)
12. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA (2003)
13. Larrea, M.L., Castro, S.M., Bjerg, E.A.: A software solution for point counting. petrographic thin section analysis as a case study. Arabian Journal of Geosciences **7**(8), 2981–2989 (2014). doi:10.1007/s12517-013-1032-0