ISBN 978-987-4417-90-9

TAPIR: An Object-Oriented Programming Testing Framework based on Message Sequence Specification with Aspect-Oriented Programming

Martín L. Larrea 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, dku}@cs.uns.edu.ar
http://vyglab.cs.uns.edu.ar

Abstract. Today we can see a significant increase in testing tools available for both the developer and the tester. Those tools aim at the testing of system implementation and vary according to the implementation paradigm, the programming language or the type of errors they seek to detect. In this paper, we present TAPIR, a white-box testing framework for Object-Oriented Programming. It was designed and implemented to detect failures in the sequence of calls that objects make. In that sense, we rely on Message Sequence Specification and Aspect-Oriented Programming. Hence, TAPIR can be used in any Java project without the need to modify the implementation of such a project. Our framework is open source and is freely available.

Keywords: Verification and Validation, Testing, Message Sequence Specification, Aspect-Oriented Programming

1 Introduction

In 1994, Kirani and Tsai [2] 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 MSS associated with an object specifies all sequences of messages that the object can receive while still providing correct behavior. In this paper, we present TAPIR a framework that implements the concept of MSS as a dynamic testing tool. The framework was designed to detect failures in the sequence of calls made by objects. In that sense, we rely on the concept of Message Sequence Specification. Its implementation was done using Aspect-Oriented Programming [3] (AOP) and, therefore, it can be used in any Java project without the need to modify the implementation of the project. The first version, 1.0, of the framework was

introduced in [7] and then 2.0 was described in [4]. Its latest version, 3.0, is presented in this paper. The change track of each version can be summarize here; Version 1.0 was limited to test only one instance of one class in the system. It was suitable only for usages where the singleton pattern was appropriate. Version 2.0 could test multiple instances of a single class in the system. Version 3.0, the one presented here, can test multiple instances of multiples classes in the system.

2 A review of Message Sequence Specification

The MSS [2] 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 method invocation. 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 $C_{M_1} \bullet C_{M_2}$. If 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 called *create*, *process*, and *close*, a possible sequencing constraint based on MSS 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 expression. If a static analysis is not enough, we could implement a runtime verification system that tracks all calls to X's methods and dynamically checks the sequence of calls against its regular expression. This technique can also be used to test the robustness of a system. Continuing with 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 method sequences. These new sequences can be used to test how the class handles a misuse. For example, how does class X respond to the following sequence of calls?: $X_{create} \bullet X_{close} \bullet X_{process}$

3 Previous Applications of Message Sequence Specification

Daniels & Tsai [1] extended the work of Kirani et al. [2] by testing with some sequences generated by the expression and others not generated by the expression. Also in 1999, Tsai et al. [9] presented Message Framework Sequence Specifications (MfSS), for generating scenario templates that can be used to generate

test cases to test applications developed using extensible design patterns and an object-oriented framework. This framework aim to test the dynamic typing and dynamic binding of an Object-Oriented program. MfSS involves with multiple objects, thus a sequence expression must use the object name together with its method name. The framework is suitable for the application of several testing technique such as positive testing, negative testing, test slicing, partition testing, boundary testing, random testing and stress testing based on scenario templates. The MSS presented in that paper is far more expressive than the one we are presenting but, as far as we know, there is no tool or software that implement this methodology. The application of the MfSS shown in the original paper was done by hand and not with a software. In 2003, Tsai [8] used MSS as a verification mechanism to the UDDI servers in the context of Web Services (WS). WS are particularly interoperative between each other. This kind of complex relationship was expressed using the MfSS presented in [9]. This approach was part of a larger set of testing mechanism for WS presented in that work. Unfortunately, only a partial part of the approach was implemented and it did not include the MfSS. In 2014, a Java-based tool for monitoring sequences of method calls was introduced [6], it had similar objectives as our work but they used annotations instead of AOP. In their work, they used annotations to specify method-call sequences in terms of regular expressions of method-call signatures. They included an implementation of the proposal, in the form of a tool called JMSeq. This tool runs alongside the application under test but in a different java virtual machine, hence there are two virtual machine running at the same time: one for the program under test and the other for the JMSeq execution. Since the annotations must be included in the source code to test, the source code must be modified, and, potentially, new errors may arise. In our approach, AOP avoids the need of modifying the source code under test, hence reducing the possibility of new bugs. We introduced MSS as a black-box technique for testing visualizations interactions [5] in 2018. The technique is built on constraints imposed over the sequences of low-level interactions available in the visualization with User Action Notation and MSS. Instead of specifying a sequence constraint on the methods of a class, in this work we specify a Sequence Constraint on the Interactions (SCI) available in the visualization. This research also included the definition of coverage criteria for the technique, both for valid and invalid sequences. The work aims to generate two types of testing tools: one to dynamically test the correct usage of a visualization tool, i.e., while the user is using the visualization our method checks that interactions are being used accordingly to the SCI; and the other, to generate the test cases based on the SCI. So far, this research is only available as theory without a proper implementation. Turner used sequence specification for GUI testing in her Ph.D. thesis in 2019, with previous publications about this topic [10], [11]. In these articles, interaction sequences are used as an abstraction of the interactive system to inform a model-based testing approach using lightweight formal methods. Interaction sequences provide an abstract view of the point at which the functional and interactive components

intersect. The formalization of interaction sequences and the modelling of those sequences was done using Finite State Automata.

4 TAPIR - An Object-Oriented Programming Testing Framework

TAPIR is a testing framework for object-oriented source code based on MSS using AOP. AOP allows us to create test cases without modifying the source code, and those test cases run automatically for every execution of the program under test. The use of MSS allows the developer to describe a regular expression for each class which represents its correct behavior. The framework takes each of these expressions, runs the program and checks that the methods are used according to its class specification. TAPIR can be classified as a dynamic analysis testing tool. A major feature of our framework is to be easy to use, with an easy to read and understand representation of the correct usage of each classes methods. Particularly, the framework was designed to be used by the developer, without the need of a testing specialist. The first thing the developer must do to use the framework is to create the regular expressions associated with the classes under test. These regular expressions must specify the correct behavior or order in which the methods of the classes should be called. In order to express this in a simple way, the developer must use symbols (i.e. characters) to represent each method. This means that the actual names of the methods are not used in the expressions. 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. Any method not included in the class's regular expression is ignore by the framework. Hence, the developer is not required to use all the class's methods in the regular expression. The developer must also specify how he/she wants the framework to behave in the event of an error. When the framework detects a sequence of calls that is not derived from its associated regular expression, it reports the error and can abort the execution or allow it to continue. This decision is in the hands of the developer and it can be specified independently for each defined regular expression. The regular expressions and the maps 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 core. Listing 1.1 shows an example with two classes: CA and CB. In this case, the correct order to use the CA class is: first, the object must be created. Then, there should be a call to *f()* followed by a call to *g()*. After that, there can be as many call as desired to either *g()* or *h()*. The final call of the sequence must be to *h()*. For the CB class, the correct use is: first, there should be a call to *alpha()* followed by a call to *gamma()* or, a call to *gamma()* follow by a call to *beta()*. Afterward, any method between alpha, beta or gamma can be called. The listing 1.1 shows how this information is input into TAPIR in the *TestingSetup.java* class.

Listing 1.1. TAPIR configuration

```
//Testing setup for CA class
//Definition of the methods and their corresponding symbols
mapObjectsToCallSequence = new HashMap<>();
mapMethodsToSymbols = new HashMap<String, String>();
mapMethodsToSymbols.put("main.CA.<init>", "c"); mapMethodsToSymbols.put("main
    .CA.f", "f"); mapMethodsToSymbols.put("main.CA.g", "g");
    mapMethodsToSymbols.put("main.CA.h", "h");
//Definition of the regular expression
regularExpression = Pattern.compile("cfg(g|h)*h");
//Initializing the regular expressions controller
matcher = regularExpression.matcher("");
// All information related to how the class is tested is store in a TestingInformation
    instance
TestingInformation ti = new TestingInformation(CA.class.toString(),
    mapObjectsToCallSequence, mapMethodsToSymbols, regularExpression, matcher,
    true);
TestingCore.mapClassToTestingInformation.put(CA.class.toString(), ti);
//Testing setup for CB class
//Definition of the methods and their corresponding symbols
mapObjectsToCallSequence = new HashMap<>();
mapMethodsToSymbols = new HashMap<String, String>(); mapMethodsToSymbols.
    put("main.CB.alpha", "a"); mapMethodsToSymbols.put("main.CB.gamma", "g");
    mapMethodsToSymbols.put("main.CB.beta", "b");
//Definition of the regular expression
regularExpression = Pattern.compile("(ag|gb)(a|g|b)*");
//Initializing the regular expressions controller
matcher = regularExpression.matcher("");
// All information related to how the class is tested is store in a TestingInformation
    instance
ti = new TestingInformation(CB.class.toString(), mapObjectsToCallSequence,
    mapMethodsToSymbols, regularExpression, matcher, false);
TestingCore.mapClassToTestingInformation.put(CB.class.toString(), ti);
```

In Listing 1.2, we can see the framework output when the code portion of Listing 1.1 corresponding to the CA class is executed. In this case, the last call to *f()* does not follow the MSS specified for the *CA* class. As mentioned above, when an error is detected, TAPIR informs this by console indicating the class and object that produced the error. The method that violated the MSS, the MSS in question and the actual sequence of calls are also shown in the console. Finally, the system abort the execution because this is what the last true parameter of method *TestingInformation* call indicates.

Listing 1.2. Error example for the CA class. The execution is aborted when the error is found.

```
CA ca1 = new CA();
ca1.f();
ca1.g();
ca1.h();
```

```
ca1.f();

--- ERROR FOUND ---
Class: class main.CA
Object Code: 977993101
Method Executed: main.CA.f
Regular Expression: cfg(g|h)*h
Execution Sequence: cfghf
----- SYSTEM ABORTING... -----
```

Listing 1.3. Error example for the CB class. The execution is allow to continue when the error is found.

```
CB cb1 = new CB();
cb1.alpha();
cb1.alpha();
cb1.gamma();
cb1.gamma();

--- ERROR FOUND ---
Class: class main.CB
Object Code: 859417998
Method Executed: main.CB.alpha
Regular Expression: (ag|gb)(a|g|b)*
Execution Sequence: aa
-- CONTINUING EXECUTION... ---
--- ERROR FOUND ---
Class: class main.CB
Object Code: 859417998
Method Executed: main.CB.gamma
Regular Expression: (ag|gb)(a|g|b)*
Execution Sequence: aag
-- CONTINUING EXECUTION... ---
```

Listing 1.2 shows the framework output when the code portion of Listing 1.1 corresponding to the CB class is executed. In this case, the second call to *alpha*() does not follow the MSS specification for class *CB*. As configured in Listing 1.1, the last false parameter in the call to method *TestingInformation* indicates that the execution must continue despite the existing errors. This is why, Listing 1.3 shows multiple errors. The next section shows how the framework can be useful in a more complex real-life situation.

5 Case Study. Earth Defender

Earth Defender is a video game developed in one of our programming courses at the university. It's a classic vertical shooter as Space Invader but with more modern features. It has 28000 lines of code distributed in 217 classes. It was developed in Java 1.8 with Eclipse IDE 4.4. There are three classes that are important for our case study; the *PlayerInteractionMananger* class is responsible

for capturing the user interactions that control the space ship's movements and shots. The *Enemy* class's responsibility is to manage each enemy ship in the game; in particular, how each ship return a power up when destroyed. Finally, the *GUI_Game* class deals with all the graphical user interfaces of the game. Because of the limitation of the number of pages in this congress, we are not showing how the $TestingSetup$ class should be configure for this case. We are only focusing on the regular expressions and the errors founds.

For the *PlayerInteractionMananger* class, the three main methods to test are *playerStartMove*, *playerStopMove* and *playerShoot*. Its MSS is $((a|b)^*x)^*$, where a stands for *playerStartMove*, b for *playerStopMove* and x for *playerShoot*. This means that, under a correct behavior, the player can move several times before shooting. This can be repeated multiple times during a game session. Two methods are considered for the*Enemy* class's MSS, those are *takeDamage* represented with a t and *dropPowerUp*, with a d. The MSS for this class is t^*d; this means that each enemy in the game can take a lot of damage until it drops its power up. Power up are dropped when enemies are destroyed. Finally, the *GUI_Game* class has the more complex MSS. Six methods are used: *inicializar* starts the GUI game level, *initLifeBar* draws the life bar on screen, *initScore* draws the initial score of the player on screen, *shoot* draws the shooting effects on screen, *changeLevel* starts the next level once the player completes the current one and, finally, *stopGame* deals with the player's death. These methods are represented by the symbols a, b, c, h, i, and j, respectively. The MSS for the *GUI_Game* class is $((bca(h)^*)i)^*(bca(h)^*)j$. The $(bca(h)^*)$ part of the MSS represents the user playing a level. If the user wins the level, then he/she moves to the next level. This is shown in $((bca(h)^*)i)^*$. The game continues until the players dies, which is the second part of the MSS $(bca(h)^*)j$.

5.1 Testing Earth Defender

We configured TAPIR to stop the execution of the game only when an error is encounter on the *GUI_Game* class. If errors are founds on the other two classes, TAPIR will inform them but allow the execution to continue. On the first test run of the game, we played the first level and lost. When we finished playing we found that TAPIR had reported three errors. The output of the framework can be seen in Listing 1.4. The three errors correspond to three different objects of the *Enemy* class. We can affirm that they are different objects because the object code that is reported in each error is different. In all three cases, by observing the execution sequence, we can detect that once a ship delivers its power-up, meaning it was destroyed, it still takes new damage. This should not happen since the ship was already destroyed. An analysis of the code involved with this situation showed that the problem was how the enemies were removed from the game once destroyed. Each time an enemy's life reached 0, the power-up was first delivered, then the score and other features of the game were updated and, finally, the enemy was removed from the level. Since the instance is eliminated last, any other shot that was on the way could still impact it. Consequently, the code was modified to remove the enemy immediately after its life reached zero.

Listing 1.4. TAPIR output of the first test run of Earth Defender

```
––– ERROR FOUND –––
Class: class Entity.Enemy
Object Code: 1256786520
Method Executed: Entity.Enemy.takeDamage
Regular Expression: t*d
Execution Sequence: tttdt
–– CONTINUING EXECUTION... –––
––– ERROR FOUND –––
Class: class Entity.Enemy
Object Code: 1337065869
Method Executed: Entity.Enemy.takeDamage
Regular Expression: t*d
Execution Sequence: ttttdt
–– CONTINUING EXECUTION... –––
––– ERROR FOUND –––
Class: class Entity.Enemy
Object Code: 1128333601
Method Executed: Entity.Enemy.takeDamage
Regular Expression: t*d
Execution Sequence: tttttttdt
–– CONTINUING EXECUTION... –––
```

We successfully completed the first level, however, since TAPIR detected errors in the *GUI_class* and it was configured to abort execution in that case, the game closed abruptly at the end of the level. The output of TAPIR can be seen in Listing 1.5. Two errors were found for the same instance of the *GUI_Game* class. In this case we can see that the object code reported are the same. We believe that TAPIR reported two errors that should abort the execution because while processing the first error, the second error was generated. In the first error, the method that causes the error is *initLifeBar*, represented by the symbol *b*. The last shot of the player can lead to two possible actions later, if with that last shot he won the level then the *changeLevel* method must be executed; on the other hand if after that last shot the player dies then the *stopLevel* method must be executed. As can be seen in the first error report detected by TAPIR, after the last shot that is represented in the sequence with *h* the method *initLifeBar* was executed. As we know that we successfully completed the level, the correct thing would have been the execution of the method *changeLevel*, symbol *i*. The second error reported is a consequence of the first, so correcting the first should fix the second. After inspecting the code, it was discovered that the method *initLifeBar* was called twice each time a level ended. This error was not noticeable from the graphical interface. One of the calls was made after changing levels, this call was correct; the second call was made as part of the logic that prepares the level change. This line of code should have been deleted in a game update but it was not. By removing this line, both errors detected by TAPIR were corrected. Three more executions of the game were made after this and the framework reported no errors.

Listing 1.5. TAPIR output on the first test run of Earth Defender

```
--- ERROR FOUND ---
Class: class GUI.GUI_Game
Object Code: 103210007
Method Executed: GUI.GUI_Game.initLifeBar
Regular Expression: ((bca(h)*)i)*(bca(h)*)j
Execution Sequence:
bcahhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhb
------ SYSTEM ABORTING... ------
--- ERROR FOUND ---
Class: class GUI.GUI_Game
Object Code: 103210007
Method Executed: GUI.GUI_Game.shoot
Regular Expression: ((bca(h)*)i)*(bca(h)*)j
Execution Sequence:
bcahhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhbh
------ SYSTEM ABORTING... ------
```

6 Conclusions & Future Work

As previously mentioned, our goal was to create an easy-to-use framework for developers to test their Java source code without any modification to it, hence reducing the possibilities of introducing new errors and minimizing preparation time. The objective of this framework is to find those vulnerabilities that allow breaking the expected order of execution of the methods of a class. As shown in the case study, the framework helps detect errors that would otherwise 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 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. In addition, we will look for the possibility of defining a MSS that involves methods of different classes; i.e., being able to combine methods of different classes in a common MSS. We will also begin to translate TAPIR into other programming languages, such as C# and Python. The framework is available for downloading[1]. The source code is available and licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Acknowledgment

This work was partially supported by the following research projects: PGI 24/N050 and PGI 24/ZN35 from the Secretaría General de Ciencia y Tecnología, Universidad Nacional del Sur, Argentina.

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

References

1. FJ Daniels and KC Tai. 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)*, pages 74–83. IEEE, 1999. doi:10.1109/TOOLS.1999.787537.
2. Shekhar Kirani and W. T. Tsai. Specification and verification of object-oriented programs. Technical report, Computer Science Department, University of Minnesota, 1994.
3. Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.
4. Martín L Larrea, Juan Ignacio Rodríguez Silva, Matías N Selzer, and Dana K Urribarri. White-box testing framework for object-oriented programming. an approach based on message sequence specification and aspect oriented programming. In *Argentine Congress of Computer Science*, pages 143–156. Springer, 2018.
5. Martín Leonardo Larrea. Black-box testing technique for information visualization. sequencing constraints with low-level interactions. *Journal of Computer Science & Technology*, 17, 2017.
6. Behrooz Nobakht, Frank S de Boer, Marcello M Bonsangue, Stijn de Gouw, and Mohammad Mahdi Jaghoori. Monitoring method call sequences using annotations. *Science of Computer Programming*, 94:362–378, 2014. doi:10.1016/j.scico.2013.11.030.
7. Juan Ignacio Rodríguez Silva and Martín Larrea. White-box testing framework for object-oriented programming based on message sequence specification. In *XXIV Congreso Argentino de Ciencias de la Computación (Tandil, 2018).*, pages 532–541, 2018.
8. Wei-Tek Tsai, Ray Paul, Zhibin Cao, Lian Yu, and Akihiro Saimi. 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).*, pages 131–138. IEEE, 2003. doi:10.1109/WORDS.2003.1218075.
9. Weik-Tek Tsai, Yongzhong Tu, Weiguang Shao, and Ezra Ebner. 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)*, pages 166–171. IEEE, 1999. doi:10.1109/CMPSAC.1999.812695.
10. Jessica Turner, Judy Bowen, and Steve Reeves. Simulating interaction sequences. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 8. ACM, 2018.
11. Jessica Turner, Judy Bowen, and Steve Reeves. Using abstraction with interaction sequences for interactive system modelling. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 257–273. Springer, 2018.