New Testing Techniques to Evaluate the Quality of Information Visualization Implementations

Martín L. Larrea, Ph.D.[1,2,3], Dana K. Urribarri, Ph.D.[1,2,3], and M. Luján Ganuza, Ph.D.[1,2,3]

[1]Department of Computer Science and Engineering, Universidad Nacional del Sur, Argentina
[2]Computer Graphics and Visualization R&D Laboratory, Universidad Nacional del Sur – CIC Prov. De Buenos Aires, Argentina
[3]Institute for Computer Science and Engineering, Universidad Nacional del Sur, Argentina
{mll, dku, mlg}@cs.uns.edu.ar

Abstract– The use of information visualization has significantly grown thanks to Industry 4.0, and now we can see its usage in more critical sectors. In this context, the implementation of such visualizations must adhere to higher quality standards. To ensure such quality, we present a set of tools developed from a software engineering perspective, particularly from the software verification and validation area. These visualization testing tools go from one to test interactions from the user point of view to another to test their code at run time. Since the toolset is free and open-source, we believe it can be the foundational basis for future developments to expand its functionality and application domains.

Keywords—Information Visualization, Software Testing, Black-box Testing, White-box Testing.

I. INTRODUCTION

Information visualizations have found enormous dissemination in the past years. Whether desktop, web, mobile, or embedded, few applications do not include any visualization. In many cases, information visualization has become crucial to industry functioning, such as those that depend on the analysis of large data sets. Within Industry 4.0, software quality has become the most critical factor in determining the success of a product in a company [1,2,3]. Information visualization, as software, must ensure the same quality levels as any other software product [4].

The software quality is constituted in multiple ways, from the execution times, usability, and the absence of errors. For the latter, software testing offers many techniques for error detection [5], which vary according to the type of error sought, software type, and other variants [6]. Information visualization requires testing techniques designed to consider its particular characteristics and, thus, find errors that otherwise would have been more difficult or impossible to find. Software Verification and Validation is an area within software engineering. It offers concepts and techniques that combine to form new testing tools, especially tools designed to evaluate the quality of information visualization implementations.

In this work, we present the usage of black-box and white-box techniques in the context of an information visualization development in C#. This work is based on Larrea et al. previous publications; we have adapted the Java tool to test C# implementations, and present the usage of those tools to perform different tests on a C# visualization. The black-box testing is based on user interactions, while the white-box is based on methods call sequences. This work includes the implementation of all necessary tools to perform these techniques.

The rest of the article is structured as follows. The next section reviews the state-of-the-art in terms of visualization testing. In the subsequent sections, we continue with the presentation of the black-box and white-box testing tools for information visualization. We develop a case study to illustrate both kinds of testing. The case study is based on a C# tool designed for the visualization of geological data, and it exemplifies the process of finding errors with the tools and methods described in this work. The last section presents the reached conclusions and the intended future work.

II. INFORMATION VISUALIZATION TESTING

When studying the testing area in the context of information visualizations, a peculiarity that emerges is the number of visualization articles where the terms "testing", "verification", and "validation" are all used as synonyms for usability evaluations. Usability testing of visualization is a well-studied area within visualization science; it focuses on how user-intuitive the visualization is. However, various works [7,8,9] emphasize the need to assess, besides intuitiveness, whether a visualization is useful for its intended purposes, i.e. evaluate its functionality. Another term that appears when exploring this area is GUI (Graphical User Interface) testing. Banerjee et al. [10] define the term GUI testing to mean that a GUI-based application, i.e., one that has a GUI front-end, is tested solely by performing sequences of events (e.g., "click on button", "enter text", "open menu") on GUI widgets (e.g., "button", "text-field", "pull-down menu"). Banerjee et al. also provide a study of the existing body of knowledge on GUI testing since 1991 and, as Memon and Nguyen [11], present a classification based on model-based GUI test techniques [12]. Hellman et al. [13] present a review of test-driven development of GUI. They state that GUI testing is very complex due partially to the degree of freedom GUIs allow users.

Those techniques which do not involve graphic components use decision tables [14] or other forms of tabular representation to test the software. Some of them are informal techniques that are very difficult to methodize and rely heavily on the tester's goodwill. Others allow systematizing the testing by using a formal specification which is very complicated to achieve for information visualization [15].

When considering these three types of testing mentioned, usability, GUI, and visualizations, we must highlight why one

cannot supplant the others. First, we must separate the usability tests from the other two. Usability testing does not test the correctness of the program [16] but whether the user can work correctly and conveniently with it. GUI and visualization testing focuses on the functionality and correctness of the system but not its usability. Regarding GUI and visualization testing, as indicated by Banerjee et. al [10], GUI Testing deals with exercising the GUI's widgets. This conception makes sense because a GUI is described in terms of widgets, such as buttons, text fields, and drop-down lists, among others. But a visualization, particularly information visualization [17], is described in terms of the abstract data it represents. Information visualization is a more abstract visual representation than GUIs and therefore requires a different approach.

In addition, some works deal with functional testing of information visualizations. Kazmi et al. [18] present what they call a meta-model for automated black-box testing of visualizations. The proposed meta-model works as the architecture of an automated testing system for visualizations; however, the authors do not present an implementation of a software for this purpose, and it is not possible to validate the model without at least one implementation. Because the proposal is a meta-model, the article does not delve into specifics of the software verification and validation areas, such as testing techniques or coverage criteria. Anbo et al. [19] focus on the research of automated testing methods for the quality of cartographic visualization to test the visualization quality of vector maps. In this context, the authors refer to quality as the union of factors that compose the quality of cartography. Although it is a broader vision than the one chosen for this work, the authors test the visualizations considering them as black boxes. Unlike Kazmi et al.'s work, the latter presents a case study on a particular map; however, the publication does not contain the set of rules used for the testing or how they were processed. Nor can it be understood from this test case how users' interactions affect the testing process. Kirby and Silva [4] highlight the need to introduce verification and validation processes for visualization development and the lack of research in this field within the visualization area. Larrea [20] also validates this statement.

Mendoça et al. [2] present a data generator application for testing visualization techniques. Their system allows users to define and compose known statistical distributions to produce the desired outcome, visualizing the behavior of the data in real-time to analyze if it has the characteristics needed for testing. They claim that testing a visualization technique with real data is very difficult; therefore, they proposed to generate these data under control conditions. They introduced a synthetic dataset generator, which the tester can use to create data sets. The tester controls the data set's characteristics such as patterns, trends, type, format, outliers, dimensions, or missing values. Although this is not a functional or usability testing technique, generating data is a fundamental step for any type of testing. Etiene's work [22,23] on verifiable visualization checks the mathematical calculations involved in the visualization process. He stated that

scientific volume rendering is not under the same rigorous scrutiny as other elements like mathematical modeling and numerical simulation. These works are more related to a white-box testing technique approach.

Motivated by the need to ensure the quality of visualizations, Larrea [20] proposed a new black-box testing technique based on user interactions at a conceptual level. Subsequent works present the creation and evolution of a white-box testing tool for Java source code [41,35,24,36] and a black-box testing tool evaluated on web-visualization test cases [40]. All these proposals build on Message Sequence Specification (MSS) [25] and coverage criteria based on method sequencing constraints [26] and share some similarities with User Action Notation [27].

III. BLACK-BOX TESTING BASED ON USER INTERACTIONS

A visualization system can be viewed as a two-part system [28], representation and interaction. The representation component is concerned with the mapping from data to visual elements and how that visual elements are rendered. The interaction component involves the dialog between the user and the system since the user expresses orders to the visualization through interactions. Interactions function as a language and, as such, may have restrictions or rules on their usage for the visualization to work properly.

A. Sequence Constraint on the Interactions

We can distinguish between high and low levels of interactions in visualization. The user objectives and the motivations behind the visualization are described in high-level interactions. The visualization's low-level interactions [29] help the user accomplish their objective, which is a high-level interaction. While high-level interactions [30,3] are more abstract, low-level interactions are present at the visualization's implementation level.

Larrea [20] introduced the concept of Sequence Constraint on Low-Level Interactions (SCI). Each SCI involves a set of binary or unary operators and a set of symbols. Each symbol represents an interaction actually available in the visualization. The SCI is a regular expression formed by those symbols and indicates the correct visualization usage.

Sequence relationships between two interactions are classified into three categories: sequential, optional, and repetition. Let V_i be the interaction i of the visualization V. Then,

- the sequential relationship $V_{i1} \bullet V_{i2}$ states that the interaction V_{i1} must be done before V_{i2},
- the optional relationship $V_{i1}|V_{i2}$ states that only one of the interactions V_{i1} and V_{i2} can be performed,
- the repetition relationship $(V_{i1})^*$ states that the interaction V_{i1} can be done many times in a row, including zero times. The + operator restricts the repetitions relationship $(V_{i1})^+$ to at least one time.

From the SCI, it can be inferred sequences of interactions that serve as test cases for the visualization. Within the presented work, we consider two types of test cases: valid test cases, i.e., sequences of interaction derivable from the SCI, and invalid test cases, i.e., interaction sequences not derivable from the SCI.

Let us take, for example, the visualization "Daily confirmed COVID-19 deaths, rolling 7-day average" [32] (Fig. 1) from Nov 1, 2021, found in "Our World in Data". Here, the user finds a visualization of a world map where each country is colored according to a color scale that is presented as part of the visualization.
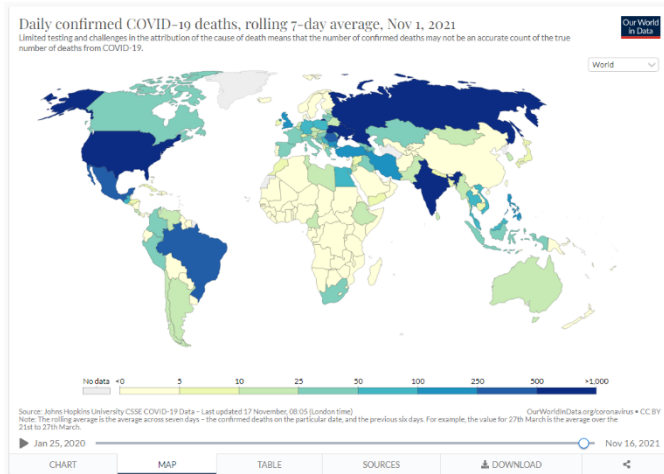

Fig. 1 Visualization about daily confirmed COVID-19 deaths, rolling 7-day average, from Nov 1, 2021. Available at [32]

The color scale shows the number of deaths caused by Covid-19. From the point of view of interactions, the user can
- Place the mouse over any point in the color scale. Consequently, the visualization highlights those countries whose color corresponds to the one located below the mouse. We call this interaction *HoverOverColorScale*.
- Hover the mouse over any country to make it stand out. The color is also highlighted on the scale. An information message also appears, inviting the user to click to access more information. We call this interaction *HoverOverCountry*.
- Click on any country, which changes the display to a line graph. We call this interaction *ClickOnCountry*.
- Once on the line chart, hover over any position on the line for additional information. We call this interaction *HoverOverLine*.
- Finally, return to the map visualization by clicking where it says Map. We call this interaction *ReturnToMap*.
- In both the map and the line graph visualizations, adjust a time slider to change the date to visualize. We call this interaction *AdjustTime-Slider*.

There are other possible interactions in this visualization, but, to explain the technique, we have limited ourselves to those presented so far. The aforementioned interactions are available to the user but cannot be used in any order. There are restrictions on when to use each one. These restrictions can be described as follows: *Once the web page loads, the user is presented with the map visualization. At this point, the user can hover the mouse over any country or the color scale or adjust the time slider. The user can click on any country, implying a previous hover over it. The line graph for a country is only accessible by clicking on that country. Once the line graph is enabled, it is possible to place the cursor on it or return to the map display. The time-slider adjustment is possible at any moment and on either of the two mentioned visualizations.*

In this functional description of the behavior of the visualization, restrictions between the interactions are evident and translatable into an SCI, as previously defined. In this sense, the SCI corresponding to this visualization would be:

$$(HoverOverColorScale \mid HoverOverCountry$$
$$\mid AdjustTimeSlider \mid HoverOverCountry \bullet ClickOnCountr \bullet$$
$$(HoverOverLine \mid AdjustTimeSlider)^* \bullet ReturnToMap)^*$$

$$(1)$$

As mentioned before, this SCI can be used to generate valid interaction sequences and infer invalid ones, which then will be used to test the behavior of the visualization. However, and in line with the principles of software testing, this testing process must be orderly, and it must be possible to establish different criteria for test intensity. Every testing technique includes these criteria, known as coverage criteria [33], and our work is not the exception.

B. Coverage Criteria

Let I be the set of interactions available on the visualization V, and G, the SCI for V using the elements of I. Consider T to be the set of test cases where each case is a sequence of interactions in I. With these elements, we can now introduce the Coverage Criteria for Sequencing Constraints with Low-Level Interactions. These criteria are divided into two categories [26]: coverage criteria for valid sequences and coverage criteria for invalid ones.

B.I Coverage Criteria for Valid Sequences

Base Coverage: Let i be the minimum length of valid sequences derived from G, then T satisfies the Base Coverage Criteria if and only if T contains all the possible i-length sequences derivable from G. If i equals 0, then T is the empty set and satisfies the Base Coverage Criteria.

Base+1 Coverage: Let i be the minimum length of valid sequences derived from G, then T satisfies the Base+1 Coverage Criteria if and only if T contains all the possible $i + 1$-length sequences derivable from G.

Base+n Coverage: This is a generalization of the previous coverage criteria.

Let i be the minimum length of valid sequences derived from G, then T satisfies the Base+n Coverage Criteria if and

only if T contains all the possible $(i + n)$-length sequences derivable from G, where $n \geq 2$. It is important to note that G may impose limits on how large n can be.

B.II Coverage Criteria for Invalid Sequences

Invalid Coverage: T satisfies the Invalid Coverage Criteria if and only if T contains all the possible 1-length sequences that are not derivable from G.

Invalid-*2* Coverage: T satisfies the Invalid-2 Coverage Criteria if and only if T contains all the possible non-derivable from G sequences obtained by combining two interactions of I.

Invalid-*n* Coverage: T satisfies the Invalid-*n* Coverage Criteria if and only if T contains all the possible non-derivable from G sequences obtained by combining n interactions of I, where $n \geq 2$.

B.III Sequence Generation for COVID-19 Visualization

From the Equation 1 and the defined coverage criteria for our technique, we can generate test sequences for the visualization of COVID-19 deaths. Starting with the valid sequences, the Base Coverage Criteria for the SCI in equation 1 is 0. Hence, the empty set satisfies the Base Coverage Criteria. For the Base+1 Coverage, we must consider valid sequences of length 1, because the base value is 0. The set $T_{valid+1}$ from Equation 2 satisfies the Base+1 Coverage Criteria because it contains all the valid sequences of length 1. Analogously, the set $T_{valid+2}$ from Equation 2 meets the Base+2 Coverage Criteria since it has all the valid sequences of length 2.

$$T_{valid+1} = \{HoverOverScale, HoverOverCountry, \quad (2)$$
$$AdjustTimeSlider\}$$

$$T_{valid+2} = \{HoverOverColorScale \quad (3)$$
$$\bullet\ HoverOverColorScale,$$
$$HoverOverCountry \bullet HoverOverCountry,$$
$$AdjustTimeSlider \bullet AdjustTimeSlider,$$
$$HoverOverColorScale \bullet HoverOverCountry,$$
$$HoverOverColorScale \bullet AdjustTimeSlider,$$
$$HoverOverCountry \bullet HoverOverColorScale,$$
$$HoverOverCountry \bullet AdjustTimeSlider,$$
$$AdjustTimeSlider \bullet HoverOverColorScale,$$
$$AdjustTimeSlider \bullet HoverOVerCountry\}$$

Each element of the sets $T_{valid+1}$ and $T_{valid+2}$ represents a test case. For every test case, a tester or the visualization developer himself must carry out the sequence of interactions and validate whether the behavior observed in the visualization corresponds to the expected one. Otherwise, a bug was detected.

A similar exercise could be done for invalid sequences following the coverage criteria. It worth noting that, in this case, if an invalid sequence can be executed in the visualization then that is the proof that there is a bug in the system. Certainly, this example allows the reader to observe that the generation of test cases is not trivial and, according to the SCI, the number of test cases can rapidly grow even for small values of coverage criteria. For this reason, this technique would benefit from tools that automate the test-case generation.

C. Test Cases Generation

A tool was developed (Fig. 2) that automates the generation of test cases to facilitate this process. From the SCI and the coverage criteria, the system automatically generates the list of test cases in a friendly format for the testing process.

This tool offers an editor that allows the user to enter the SCI, the coverage parameters, and an optional mapping between symbols and interaction names. To simplify the writing of the SCI, the tool allows the user to use single characters as interactions. The user can then map these characters to meaningful interaction names to make the test cases easier to read.

Once the information is entered and validated, the tool generates a report containing the test cases. The report design allows recording the success or failure of each test case and attaching notes of the execution. In addition, it can be used on the web or as a printed document.

Fig. 2 shows a screenshot of the tool in its initial state. As seen in the figure, it is composed by three mandatory input fields in which the user enters the regular expression and the two values used as parameters for the coverage criteria. These last two are initialized by default with the values 0 and 1 since they are the minimum values allowed by definition. In turn, the input fields do not allow entering smaller values. Once a valid SCI expression has been entered, new fields are dynamically generated for the user to optionally enter the full name of the interaction, as shown in Fig. 3. Symbol mapping is optional at the individual level. It allows adding a more descriptive name only for those symbols the user considers worthwhile.

When viewing the report, the names in the mapping are used to display a detailed version of the SCI expression and list the interaction names in each test case. In case an error is detected in the entered values, a descriptive message is displayed. Once the required values are entered, the user is enabled to generate the report, and the newly generated one shows up in a browser new tab. Since the report was designed and implemented with simplicity in mind, the same format presented in the application could be exported as PDF using the standard printing functionality directly. The blue PDF icon is the button that allows users to export the document as a PDF file (Fig. 4).

A heading at the top of the report shows the values previously entered in the editor and used to generate the test-cases list. The cases in the list are grouped according to whether they are a valid or invalid sequence of interactions. For each test case, there is a box with a title showing the sequence of interactions from the SCI expression that resulted in that test case. As mentioned above, names are used in those cases where a mapping was provided.

The report allows users to check whether the test case was successfully executed or failed at some point. After a test-case execution, the user can select the result in the upper right corner

and write comments if necessary. Note that a valid test case is successful if the sequence executes correctly; however, an invalid test case is successful if the sequence fails to execute.

IV. WHITE-BOX TESTING BASED ON MSS

The development of this visualization-testing technique led to the development of a second testing tool but oriented to white-box testing. This new development has similarities with another developed in the 90s [26, 34], in the sense that it works on restrictions to the order of calls to methods of a class instead of interactions of a visualization. This new tool, called TAPIR, became a testing framework for Java [35], and, in this work, we adapted it for C#.

TAPIR [41,35,24,36, 43] is a white-box testing framework for object-oriented source code based on Message Sequence Specification (MSS) [34]. An MSS is the equivalent of an SCI but for a class in an object-oriented program. It describes the correct order in which the methods of a class should be invoked by its clients.
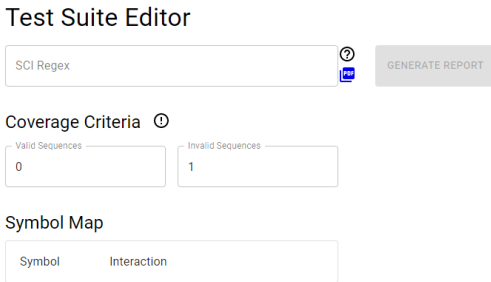

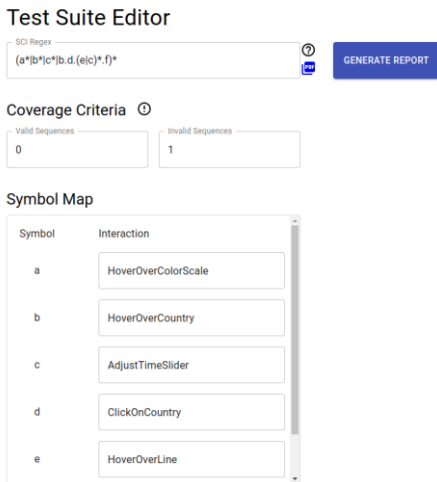Fig. 2 Home screen of the Test Suite Editor, a tool to automatically generate test cases. Available at [42].

Fig. 3 Once a valid SCI expression has been entered, new fields are dynamically generated for the user to optionally enter the interaction full-name to make the test cases easier to read.
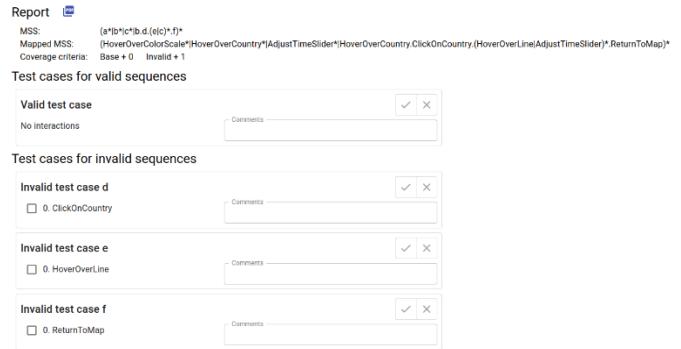

Fig. 4 A heading at the top of the report shows the values previously entered in the editor and used to generate the test-cases list. The blue PDF icon is the button that allows users to export the document as a PDF file.

Each class can have an associated MSS which specifies all sequences of messages that the class's instances can receive while still providing correct behavior. Sequence relationships between two methods are classified into three categories, like before: sequential, optional, and repetition.

A. Runtime Verification

TAPIR allows the programmer to verify, at runtime, that the sequence of method calls of an object respects the MSS defined for the class of which the object is an instance. The framework now allows the incorporation of these controls into any Java or C# code without having to modify it. This is achieved thanks to the fact that the TAPIR core is implemented using Aspect-Oriented Programming. The framework consists of two main components: an aspect and two classes. The aspect is named *TestingCore* and contains the implementation of the framework's core. The two classes are *TestingInformation* and *TestingSetup*.

Class *TestingInformation* encapsulates all the information necessary to test a particular class, namely:

- a map between class instances and the actual invocation sequences for each one,
- the regular expression that describes the MSS of the class,
- a map between the name of the class's methods and the symbols used in the regular expression,
- whether the execution of the program should abort after an error in the invocation sequence of this class.

The *TestingSetup* class is responsible for initializing the framework with the data about the classes to test. This initialization is responsibility of the developer and must be implemented in a *setup* method. The *TestingCore* aspect, before the execution of the *main* method, initializes the framework executing method setup from class *TestingSetup*. Then, the aspect captures each method call that occurs outside the framework scope and checks whether it corresponds to a class under testing. In that case, the framework tests whether the method was invoked following the regular expression in its class specification. The TAPIR framework was implemented in

Java using the AspectJ extension [37]. We are now introducing its version for C# which was implemented with PostSharp [38].

Take for example a class that implements a checking account, as shown in Listing 1. Let us then consider the following conditions for the correct use of this class; first, the account must be created, and then it must be verified. Next, the first movement of money must be a deposit so that afterward the user can deposit or withdraw money as many times as she/he wants. Once the account is closed, no further operations are allowed. Based on this specification we can establish the following MSS for the *CheckAccount* class:

$$create \bullet verify \bullet deposit \bullet (deposit \mid withdraw)^* \bullet close \qquad (4)$$

Listing 1 A simple class to demonstrate how the TAPIR framework works in C#.

```
class CheckAccount {
    private int amount;
    private Boolean verified;

    public CheckAccount() {
        amount = o;
        verified = false ; }

    public void Verify() {
        verified = true; }

    public Boolean IsVerified() {
        return verified ; }

    public void Deposit(int amount) {
        if ( IsVerified () ) {
            amount += amount; } }

    public void Withdraw(int amount) {
        if ( IsVerified () ) {
            amount -= amount; } }

    public void Close() {
        amount = o;
        verified = false ; } }
```

The first thing the developer must do to use the framework is to identify and write the MSS associated with the classes under test. To simplify its writing, symbols (i. e., characters) are used instead of the actual names of the methods. However, to be able to interpret it, the developer must specify a mapping between the actual methods' names and their corresponding symbol. TAPIR ignores any method not included in the class's MSS. Hence, the developer is not required to use all the class methods in the MSS. Implicitly, leaving a method out of the SCI indicates that that method can be invoked at any time (for example, the *IsVerified()* method of class *CheckAccount*.

The developer must also specify how the framework should behave in the event of an error. When TAPIR detects a sequence of calls that do not derive from its associated MSS, it reports the error and can either 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 MSS. The

regular expressions and the maps between methods and symbols are set in the *TestingSetup* class. Listing 2 shows this for the *CheckAccount* class.

Listing 4 shows the framework output of the execution of the code portion of Listing 3, which corresponds to a misuse of the *CheckAccount* class. In this case, the call to *Verify()* does not follow the MSS specified for the *CheckAccount* class. When an error is detected, TAPIR reports to the console the object that caused the error, the class of which it is an instance, the method that violated the MSS, the MSS in question, and the actual sequence of calls. Finally, the system aborts the execution as indicated in the configuration by the last parameter at the constructor invocation of the *TestingInformation* (see Listing 2). Each *TestingInformation* instance corresponds to the MSS and the symbol mapping for one class. Then, adding more instances to the *TestingCore* class allows the testing of multiple instances of multiple classes.

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 the class' methods. In particular, the user of the framework does not need to be a testing specialist since it was developed to be used directly by the developer.

B.   Test Cases Generation

Larrea & Urribarri [36, 44] presented Generotron (Fig. 5), a complement to TAPIR for the generation of test cases and the methodical testing of the behavior of a class against valid and invalid combinations of method invocations. This complement generates valuable documentation for developers when unit testing classes. Even though TAPIR evaluates the correct usage of a set of classes at run-time, unit testing is still necessary when testing software components that are not yet part of a complete application. Since programming knowledge is not required to use this application and its output is language-independent, any work-team member can generate such test cases. This application is equivalent to the one developed for information visualization but with class terminology. Both offer the same functionalities for writing regular expressions (SCI and MSS) and generating reports.

V. SPINELVIZ – A VISUALIZATION APPLICATION FOR MINING AND GEOLOGICAL INDUSTRIES

In this section, we show how the presented toolset can be used to detect possible errors in an interactive 3D application for visualizing spinel-group minerals data called SpinelViz [39]. The spinel-group minerals constitute excellent petrogenetic indicators and guide the search for mineral deposits of economic interest. The application consists of an interactive 3D viewer, which allows the user to view and explore different datasets simultaneously in the same spinel prism.   Geologists usually represent the composition of the spinel-group minerals in a prismatic space called spinel prism. SpinelViz provides the capability to manipulate, view, plot, and project data in 2D and 3D, which helps the user to gain a better insight into the data distribution.

Listing 2 TAPIR configuration for verifying the invocation order of the methods of the *CheckAccount* class at runtime.

```
class TestingSetup {
    public static void Setup() {
        //Specification of the test class
        TestingCore.MapClassToTestingInformation = new Dictionary<string, TestingInformation>();
         //Testing setup for CheckAccount class
        //Definition of the methods and their corresponding symbols
        mapObjectsToCallSequence = new Dictionary<int, string>();
        mapMethodsToSymbols = new Dictionary<string, string>();
        mapMethodsToSymbols.Add("ConsoleApp1.main.CheckAccount..ctor", "c");
        mapMethodsToSymbols.Add("ConsoleApp1.main.CheckAccount.Verify", "v");
        mapMethodsToSymbols.Add("ConsoleApp1.main.CheckAccount.Deposit", "d");
        mapMethodsToSymbols.Add("ConsoleApp1.main.CheckAccount.Withdraw", "w");
        mapMethodsToSymbols.Add("ConsoleApp1.main.CheckAccount.Close", "x");
        //Definition of the regular expression
        potentialRegularExpression = @"^c(v(d((d|w)*x?)?)?)?$";
        finalRegularExpression = @"^cvd(d|w)*x$";
        //A TestingInformation instance stores all the information related to how is tested the CheckAccount class
        className = typeof(CheckAccount).FullName;
        testingInformation = new TestingInformation(className, mapObjectsToCallSequence, mapMethodsToSymbols, potentialRegularExpression,
finalRegularExpression, true);
        TestingCore.MapClassToTestingInformation.Add(className, testingInformation);
    }
}
```

A. Black-box Testing based on User-Interactions Tool

A common task in the spinel-mineral analysis is to analyze a particular sample (data item) in the context of the dataset to which it belongs. For this reason, SpinelViz supports on-demand selection and highlighting of particular samples. After loading a dataset, the SpinelViz represents all the data items corresponding to that dataset with the same color, shape, and size. Then, the user can select a data item of interest and modify its representation to differentiate it from the rest.

This modification can be undone, restoring the data item to its original visual appearance. A list in the interface registers all the currently modified items. Undoing a modification must remove the item from the list.

Listing 3 Example of a misuse of the methods of class *CheckAccount*.

```
var account9 = new CheckAccount();
account9.Verify();
account9.Deposit(1000);
account9.Deposit(4000);
account9.Withdraw(3000);
account9.Verify();
account9.Close();
```

Listing 4 Error example for the *CheckAccount* class. The execution is aborted when the error is found

```
---     ERROR FOUND     ---
Class: class main.CheckAccount
Object Code: 1421795058
Method Executed: main.CheckAccount.Verify
Regular Expression: cvd(d|w)*x
Execution Sequence: cvddwv
----- SYSTEM ABORTING... -----
```

For this matter, the application supports the following interactions:

- Load a dataset into the spinel prism. All the data items of the loaded dataset are represented in the spinel prism with the same color, shape, and size. We call this interaction *Load*.
- Select a data item. We call this interaction *Select*.
- Modify the visual representation of a selected sample. The user can change the color, size, and shape used to represent the data item in order to highlight it. We call this interaction *Modify*.
- Undo the modifications performed over a sample. The user can undo the changes on a selected sample and restore it to its original visual appearance. We call this interaction *Undo*.

The SCI of Equation 5 describes the behavior of this visualization. Once the SpinelViz loads, the user is presented with an empty spinel prism. To start the analysis session he/she must *Load* a dataset. After loading the dataset, the user can *Select* a data item and *Modify* its representation. Then the user can *Select* any data item and *Modify* it or *Undo* the modification if the selected item was previously modified. The list of currently modified items should change accordingly.

$$Load \mid (Load \bullet Select \bullet Modify \bullet (Select \bullet (Modify|Undo))^*) \tag{5}$$

Since it is mandatory to load a dataset to start the analysis session, the minimum sequence of interactions valid for this visualization is *1*. The test set $T_{Valid+0}$ that satisfies the Base Coverage criteria contains only one sequence of one interaction (see Equation 6).

$$T_{Valid+0} = \{Load\} \tag{6}$$

From that SCI, it is impossible to derive valid sequences of length *2* or *4*, then the test sets $T_{Valid+1}$ and $T_{Valid+3}$ that satisfy

the *Base+1* and *Base+3* coverage criteria are empty. With the provided tool, we easily generated the test sets that satisfy the *Base+2* and *Base+4* Coverage criteria (see Equations 7 and 8).

$$T_{Valid+2} = \{Load \bullet Select \bullet Modify\} \qquad (7)$$

$$T_{Valid+4} = \{Load \bullet Select \bullet Modify \bullet Select \bullet Modify, \qquad (8)$$
$$Load \bullet Select \bullet Modify \bullet Select \bullet Undo\}$$

The report provided by the tool was very useful to guide the testing of the SpinelViz (see Fig. 6). While writing the SCI, *L* stands for *Load*, *S* for *Select*, *M* for *Modify*, and *U* for *Undo*. Fortunately, no errors were found when executing the interaction sequences from $T_{Valid+0}$ and $T_{Valid+2}$.
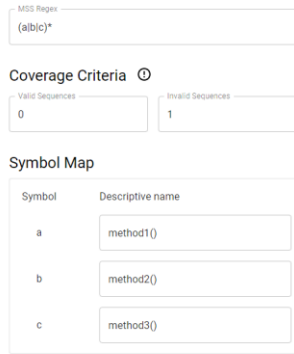

Fig. 5 Homescreen of Generotron, our application for automatic test cases generation based on MSS.
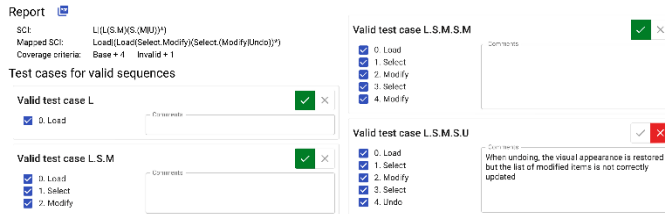

Fig. 6 Report of test cases with coverage criteria *Base+0*, *Base+2* and *Base+4* for valid sequences.
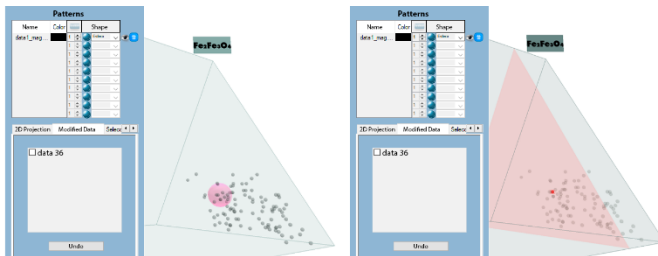

Fig. 7 Modify and Undo interactions in SpinelViz. On the left, the result of modifying the visual appearance of a data item is shown. On the right, when the user undoes the modification, the visual appearance is restored, but the list of modified items is not correctly updated.

However, the SpinelViz did not work properly for all test cases belonging to $T_{Valid+4}$. Unexpectedly, when trying to run the valid sequence {*Load* • *Select* • *Modify* • *Select* • *Undo*} we realized that the *Undo* operation did not work properly. Although the visual appearance of the selected data item was restored correctly, the list of currently modified elements was not correctly updated in the interface (Fig. 7).

B.   White-box Testing based on MSS using TAPIR

In this section, we try to determine if the error detected in Section V.A is the result of a failure in the sequence of calls made by objects. SpinelViz was developed in C#. It has 51540 lines of code distributed in 29 classes. The class strictly related to the error found in the previous section is *ModifiedItems*, a class that encapsulates a data structure responsible for maintaining a record of the currently modified data items. The primary methods to test in this class are the *constructor*, *addModifiedItem*, and *removeModifiedItem*. Equation 9 shows its corresponding MSS.

$$create \bullet addModifiedItem \qquad (9)$$
$$\bullet (addModifiedItem \mid removeModifiedItem)^*$$

We configured TAPIR with the mentioned MSS and worked with the SpinelViz during a long analysis session. Since the error detected in Section V.A had not yet been fixed, the list of modified elements was not updated correctly when the user tried to undo a modification; however, the analysis session finished, and TAPIR did not detect any failure in the sequence of calls.

At this point, we knew that something was wrong with the *ModifiedItems* class, and as TAPIR supports working with private and protected methods, we decided to continue testing the class at a lower level. The *ModifiedItems* class encapsulates a data structure that stores the modified data items, and for matters of efficiency, it maintains an index to the last element in the structure.

Then, the *addModifiedItem* method is the result of calling two protected methods, *addItem* (in charge of inserting the new item at the end of the structure) and *updateIndex* (in charge of updating the index accordingly). In the same way, the *removeModifiedItem* method is the result of calling two protected methods, *removeItem* (in charge of deleting the corresponding item from the structure) and *updateIndex* (in charge of updating the index accordingly). Therefore, Equation 10 shows the new MSS for the class at the protected methods level.

$$create \bullet addItem \bullet updateIndex \qquad (10)$$
$$\bullet ((addItem$$
$$\bullet updateIndex) \mid (removeItem$$
$$\bullet updateIndex))^*$$

We configured TAPIR to stop the execution of the application when encountering an error. At first we performed the sequence of interactions that manifested the error during the

black-box testing. We loaded a dataset, selected and modified a data item, and undid the modify operation. Since TAPIR did not report any error at that moment, we continued working and tried to modify another data item. Then TAPIR stopped the execution and reported the error of Listing 5. As we can see in that error, when removing the modified item, the indexes were not updated, causing the error detected in Section V.A.

C. White-Box Testing based on MSS using Generotron

An essential class within SpinelViz is *Prism*, which encapsulates the prism and provides methods to draw it on the screen. Class *Prism* has the following methods:

- *drawPrism*: to draw the prism on the screen.
- *drawSelection*: draw a triangle representing the cross section of the prism that passes through the selected item (see Fig. 7).

Listing 5 Error reported by TAPIR while executing the protected method *addItem* of class *ModifiedItems*.

```
---    ERROR FOUND    ---
Class: Prism.Strucures.ModifiedItems
Object Code: 35191196
Method Executed: Prism.Structures.
ModifiedItems.addItem
Regular Expression: ^cau(au|ru)*?$
Execution Sequence: caura
----- SYSTEM ABORTING... -----
```
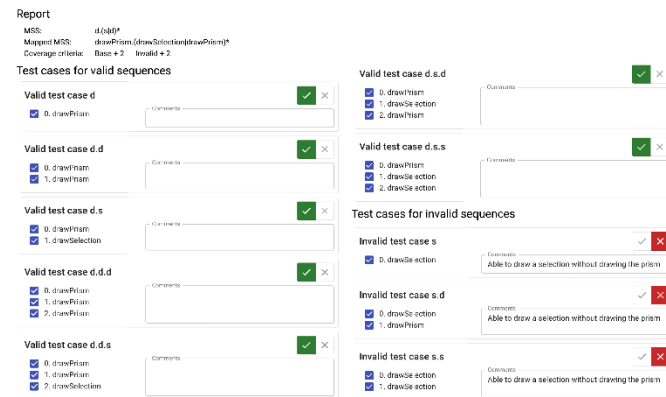

Fig. 8 Report generated with Generotron for valid and invalid sequences.

A prism can be drawn at any time after its creation. However, before drawing a selection, the prism must have been drawn at least once. Equation 11 shows the MSS that describes the correct operation of the class.

$$drawPrism \bullet (drawSelection|drawPrism)^* \qquad (11)$$

We used Generotron to generate test cases for the class with Base+2 coverage criteria for valid and invalid sequences. In the symbol mapping, *d* stands for *drawPrism* and *s* for *drawSelection*. Fortunately, all the valid sequences generated by Generotron, following the specified coverage criteria, executed successfully. However, we faced some problems when testing the invalid sequences, as the system allowed the

execution of three invalid sequences of calls without reporting any error. As shown in Fig. 8, SpinelViz allowed the execution of the invalid sequences $\{s, s \bullet d, s \bullet s\}$, meaning that the system did not report an error when a selection was drawn without drawing a prism first. Despite the fact that these sequences should not occur during normal execution, the class is expected to be robust enough to withstand misuse.

VI. CONCLUSIONS & FUTURE WORK

With the dissemination of information visualization across different type of platforms, devices, and application domains comes the need to ensure their quality in a way that has never been required before. In this context, it is essential to develop new and better methodologies and tools that allow the visualization developer to ensure the correct functioning of visual representations and interactions. Software engineering offers the basis for adapting existing testing techniques to the particularities of information visualization and also the possibility of creating new techniques.

We have presented a series of tools for testing visualization implementations from both a black-box and a white-box perspective. From a white-box point of view, we adapt the TAPIR testing framework to work with C#. The entire source code is available open-source at [43]. However, the framework lacks expressiveness in regular expressions for both testing perspectives. In particular, in the current state, it is only possible to represent actions but not conditions on those actions. Furthermore, although it is possible to automatically generate test cases from the regular expressions, the framework does not allow their automatic execution. These two framework deficiencies are the ones that need the most attention.

REFERENCES

[1] E. Oztemel and S. Gursev, "Literature review of Industry 4.0 and related technologies," Journal of Intelligent Manufacturing, vol. 31, no. 1, pp. 127–182, 2020.

[2] M. Lee, J. J. Yun, A. Pyka, D. Won, F. Kodama, G. Schiuma, H. Park, J. Jeon, K. Park, K. Jung et al., "How to respond to the fourth industrial revolution, or the second information technology revolution? Dynamic new combinations between technology, market, and society through open innovation," Journal of Open Innovation: Technology, Market, and Complexity, vol. 4, no. 3, p. 21, 2018.

[3] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," Journal of industrial information integration, vol. 6, pp. 1–10, 2017.

[4] R. M. Kirby and C. T. Silva, "The need for verifiable visualization," IEEE Computer Graphics and Applications, vol. 28, no. 5, pp. 78–83, 2008.

[5] A. Spillner and T. Linz, Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant. Dpunkt. verlag, 2021.

[6] P. C. Jorgensen, Software testing: a craftsman's approach. Auerbach Publications, 2013.

[7] M. D. Gerst, M. A. Kenney, and I. Feygina, "Improving the usability of climate indicator visualizations through diagnostic design principles," Climatic Change, vol. 166, no. 3, pp. 1–22, 2021.

[8] A. Vizoso, "Information Visualization and Usability: Tools for ´ Human Comprehension," in Journalistic Metamorphosis. Springer, 2020, pp. 85–98.

[9] D. Dowding and J. A. Merrill, "The development of heuristics for evaluation of dashboard visualizations," Applied clinical informatics, vol. 9, no. 3, p. 511, 2018.

[10] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," Information and Software Technology, vol. 55, no. 10, pp. 1679–1694, 2013.

[11] A. M. Memon and B. N. Nguyen, "Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End," in Advances in Computers, ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2010, vol. 80, pp. 121–162.

[12] I. Banerjee, "Advances in model-based testing of GUI-based software," in Advances in Computers. Elsevier, 2017, vol. 105, pp. 45–78.

[13] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer, "Agile interaction design and test-driven development of user interfaces–a literature review," Agile Software Development, pp. 185–201, 2010.

[14] S. Supriyono, "Software Testing with the approach of Blackbox Testing on the Academic Information System," IJISTECH (International Journal of Information System & Technology), vol. 3, no. 2, pp. 227–233, 2020.

[15] M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, and S. Shaikh, Formal Methods for Software Engineering: Languages, Methods, Application Domains. Springer International Publishing, 2020.

[16] S. Lauesen, "Usability engineering in industrial practice," in Human-Computer Interaction INTERACT'97. Springer, 1997, pp. 15–22.

[17] C. Ware, Information visualization: perception for design. Morgan Kaufmann, 2019.

[18] S. H. Kazmi, F. Azam, M. W. Anwar, and B. Maqbool, "A MetaModel for Automated Black-Box Testing of Visualization Based Software Applications," in Proceedings of the 2020 9th International Conference on Software and Computer Applications, 2020, pp. 183–187.

[19] A. Li, L. Hong, and J. Cao, "Study on the method of cartographic visualization quality automated testing," in 2010 18th International Conference on Geoinformatics, 2010, pp. 1–6.

[20] M. L. Larrea, "Black-box testing technique for information visualization. Sequencing constraints with low-level interactions," Journal of Computer Science & Technology, vol. 17, 2017.

[21] S. D. P. Mendonc̦a, Y. P. D. S. Brito, C. G. R. Dos Santos, R. D. A. D. Lima, T. D. O. De Araujo, and B. S. Meiguins, "Synthetic datasets ́ generator for testing information visualization and machine learning techniques and tools," IEEE Access, vol. 8, pp. 82 917–82 928, 2020.

[22] T. Etiene, D. Jonsson, T. Ropinski, C. Scheidegger, J. L. Comba, ̈ L. G. Nonato, R. M. Kirby, A. Ynnerman, and C. T. Silva, "Verifying Volume Rendering Using Discretization Error Analysis," IEEE Transactions on Visualization and Computer Graphics, vol. 20, no. 1, pp. 140–154, 2014.

[23] Etiene, et al, "Topology Verification for Isosurface Extraction," IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 6, pp. 952–965, 2011.

[24] M. L. Larrea and D. K. Urribarri, "TAPIR: An Object-Oriented Programming Testing Framework based on Message Sequence Specification with Aspect-Oriented Programming," in XXVI Congreso Argentino de Ciencias de la Computaci ́on (CACIC), 2020, pp. 389–398.

[25] S. Kirani and W. T. Tsai, "Specification and Verification of Object-Oriented Programs," University of Minnesota, Tech. Rep., 1994.

[26] F. Daniels and K. 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). IEEE, 1999, pp. 74–83.

[27] H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: A user oriented representation for direct manipulation interface designs," ACM Transactions on Information Systems (TOIS), vol. 8, no. 3, pp. 181–203, 1990.

[28] J. S. Yi, Y. ah Kang, J. Stasko, and J. A. Jacko, "Toward a deeper understanding of the role of interaction in information visualization," IEEE transactions on visualization and computer graphics, vol. 13, no. 6, pp. 1224–1231, 2007.

[29] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in The craft of information visualization. Elsevier, 2003, pp. 364–371.

[30] B. Kovalerchuk, B. Kovalerchuk, and J. Schwing, Visual and spatial analysis. Springer, 2004.

[31] P. R. Keller, M. M. Keller, S. Markel, A. J. Mallinckrodt, and S. McKay, "Visual cues: practical data visualization," Computers in Physics, vol. 8, no. 3, pp. 297–298, 1994.

[32] "Daily confirmed COVID-19 deaths, rolling 7-day average, Nov 1, 2021," https://ourworldindata.org/grapher/daily-covid-deaths-7-day?time=2021-11-01&country=~CHN, accessed: 2021-11-13.

[33] M. Friske, B.-H. Schlingloff, and S. Weißleder, "Composition of Model-based Test Coverage Criteria," in MBEES, 2008, pp. 87–94.

[34] S. H. Kirani and W. Tsai, "Specification and verification of objectoriented programs," Ph.D. dissertation, Citeseer, 1994.

[35] M. L. Larrea, J. I. Rodriguez Silva, M. N. Selzer, and D. K. Urribarri, "WhiteBox Testing Framework for Object-Oriented Programming. An approach based on Message Sequence Specification and Aspect Oriented Programming," in Argentine Congress of Computer Science. Springer, 2018, pp. 143–156.

[36] M. L. Larrea and D. K. Urribarri, "Expanding the scope of a testing framework for Industry 4.0," in XXVII Congreso Argentino de Ciencias de la Computación (CACIC). 2021.

[37] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming. Greenwich, CT, USA: Manning Publications Co., 2003.

[38] H. Bopuri and R. Salman, "Aspect Oriented Programming Through C# .NET," International Journal of Software Engineering (IJSE). CSC Journals, Kuala Lumpur Malaysia, vol. 4, no. 1, pp. 23– 32, 2013.

[39] M. L. Ganuza, S. M. Castro, G. Ferracutti, E. A. Bjerg, and S. R. Martig, "SpinelViz: An interactive 3D application for visualizing spinel group minerals," Computers & Geosciences, vol. 48, pp. 50–56, 2012.

[40] M. Schiaffino, M. L. Larrea, M. L. Ganuza, D. K. Urribarri. A Testing Tool for Information Visualizations based on User Interactions. Journal of Computer Science & Technology, vol. 22, no. 1, pp. 78–92, 2022. DOI: 10.24215/16666038.22.e06

[41] Rodríguez Silva, J. I., & Larrea, M. L. (2018). White-Box Testing Framework for Object-Oriented Programming based on Message Sequence Specification. In XXIV Congreso Argentino de Ciencias de la Computación (La Plata, 2018).

[42] Test Suite Editor, https://cs.uns.edu.ar/~dku/vis/visualization-sci-testing/editor. Accessed: 2023-05-02.

[43] TAPIR, White-box Testing Framework, https://cs.uns.edu.ar/~mll/lapaz/. Accessed: 2023-05-02.

[44] Generotron, http://www.cs.uns.edu.ar/~dku/vis/mss/editor. Accessed: 2023-05-02.