A Testing Tool for Information Visualizations based on User Interactions

Herramienta de Testing para Visualización de Información Basada en Interacciones de Usuario

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

[1]Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS), Bahía Blanca, Argentina
[2]Computer Graphics and Visualization R&D Laboratory, Universidad Nacional del Sur (UNS) - CIC Prov. Buenos Aires, Bahía Blanca, Argentina
[3]Institute for Computer Science and Engineering, Universidad Nacional del Sur (UNS) - CONICET, Bahía Blanca, Argentina
schiaffinomartin@gmail.com, {mll, mlg, dku}@cs.uns.edu.ar

Abstract

Decision-making has become a vital process in any organization, evolving from a process based on experience and intuition to one increasingly established in data analysis. One type of specialized software for data analysis is that of visual representations for large data sets. Visual representations are critically important today as they enable effective exploration of a data set and facilitate the task of identifying patterns and drawing conclusions. Every day more decisions are made based on visual analysis through visual representations of large data sets. It is not only a quantitative but also a qualitative increase. Decisions are more critical and with more impact on society, the environment, and individuals. 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 their interactions. To achieve this goal, we present Test Suite Editor, a platform that assists in visualization testing. This platform facilitates the generation of test cases based on user interactions. This contribution is based on a previously published black box testing technique for information visualizations that uses regular expressions to represent the sequence of user interactions.

Keywords: Information Visualization, User Interactions, Regular Expressions, Software Testing.

Resumen

La toma de decisiones se ha convertido en un proceso vital en cualquier organización, evolucionando de un proceso basado en la experiencia y la intuición a uno basado en el análisis de datos. La visualización es un tipo de software especializado para el análisis de grandes conjuntos de datos. La visualización de información es de vital importancia hoy en día, ya que permite la exploración efectiva de un conjunto de datos y facilita la tarea de identificar patrones y sacar conclusiones. Cada día se toman más decisiones basadas en el análisis visual a través de representaciones visuales de grandes conjuntos de datos. No es solo un aumento cuantitativo, sino también cualitativo. Las decisiones son más críticas y tienen más impacto en la sociedad, el medio ambiente y las personas. En este contexto, es fundamental desarrollar nuevas y mejores metodologías y herramientas que permitan al desarrollador de la visualización asegurar el correcto funcionamiento de las representaciones visuales y sus interacciones. Para lograr este objetivo, presentamos Test Suite Editor, una plataforma que ayuda en el testing de visualizaciones. Esta plataforma facilita la generación de casos de prueba basados en las interacciones del usuario. Esta contribución se basa en una técnica de prueba de caja negra publicada anteriormente para visualizaciones de información que utiliza expresiones regulares para representar la secuencia de interacciones del usuario.

Palabras claves: Visualización de Información, Expresiones Regulares, Testing de Software.

1 Introduction

The visual representations of information, particularly for large data sets, is an area of Computer Science for which its application in multiple fields has grown steadily for many years. These visual representations have become a fundamental tool in the analysis and decision-making processes. It was not only a quantitative increase but also a qualitative one. The decisions are more critical and with more impact on society, the environment, and individuals. For this reason, it is necessary to have the appropriate tools to ensure the correct functioning of the visual representations.

The visualizations are software products then, it is possible to evaluate them using those techniques of software testing proposed by the Software Verification

and Validation (V&V) area. Many V&V [1] techniques are applicable at different stages of software development. The two main categories are white-box and black-box techniques. In the first one, the testing is driven by the knowledge and the information provided by the software implementation, i.e. the source code. In the second one, the specification of the software, the module, or the function is used to test the software.

The source code of visualization software is just another piece of software; therefore, it can be tested with any available white-box technique [2]. All black-box testing techniques design their test cases based on the software specification. Some of these techniques involve the GUI (Graphical User Interface) components of the software and their interactions. Buttons, text fields, and drop-down lists are common elements among those GUI. Nonetheless, a visualization constitutes a GUI by itself with more components than a regular user interface. Besides buttons and text fields, a visualization may have glyphs, axes, 3D or 2D visual objects that change location or shape according to the user's interactions. In these cases, the black-box techniques that rely only on traditional GUI components are not suitable. Those techniques which do not involve graphic components use decision tables [3] or other forms of tabular representation to test the software. Some of them are very 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 [4].

In this context, we present Test Suite Editor, an easy-to-use platform designed specifically for testing information visualization with a black-box approach. Test Suite Editor automates the generation of test cases based on user interactions. Although this platform cannot execute the test cases yet, generating them reduces test times and allows for orderly testing. Our main goal is to provide a tool with an easy-to-read and easy-to-understand methodology to test each visualization. Our goal is to provide a tool that the developer, or even the visualization user, can use without needing a testing specialist. Our primary intention is to reduce the disconnection between commercial tools and literature proposals and between researchers and practitioners, as mentioned by Banerjee et. al [5]. One of our goals with this work is to be able to provide the information visualization community with a testing tool designed for them.

This paper continues with a brief introduction to Software Testing and its terminologies in Section 2 and outlines the previous work on visualization testing in Section 3. Section 4 describes sequencing constraints with low-level interactions as a testing methodology, which is the basis of the work described in this article. Sections 5 to 6 describe the proposed implementation of an information visualization testing tool and the application of this tool to two test cases. Finally, Section 7 draws some conclusions and presents possible future work.

2 Background of Software Testing

This section is intended for readers outside the Software Testing discipline. Here we present a brief introduction to those concepts needed to follow the development of this article. For this purpose, we are using the work of [1] as a reference. For a better understanding of the subject, we recommend that the reader consult the cited bibliography.

The dynamic behavior of the software is checked through testing, where the tested software will be executed. Its behavior must be compared to the given requirements. A situation can be classified as incorrect only if we know which is the expected correct situation. Thus, a failure is a discrepancy between the observed behavior and the expected one. We must differentiate between the occurrence of a failure and its cause. A failure has its root in a fault in the software, or as it is more popularly called, in a bug. Testing is not debugging; while testing is responsible for detecting failures, debugging deals with locating the bugs that caused the failure in the software.

The element that we are testing is defined as a test object. The execution of a test object is done with test data. The administration of the tests includes the planning, implementation, documentation, and analysis of the testing. A test suite is defined as the execution of one or more test cases. A test case contains the test object, execution conditions, input parameters, and expected output. The concatenation of test cases, so that the input of a test is the output of the previous one, forms a test scenario.

Several different approaches are available for testing the test object. They can be categorized into two groups: black box and white box testing. In black-box testing, the test object is seen as a black box. Test cases are derived from the specification of the test object. The behavior of the test object is watched from the outside. The operating sequence of the test object can only be influenced by choosing appropriate input test data or by setting appropriate preconditions. In white-box testing, the source code is known and used for test design. While executing the test cases, the internal processing of the test object and the output are analyzed. Both white-box and black-box testing techniques must describe a test model and, at least, one coverage criterion. A test model describes how to generate test cases, and it can be a graph, a table, or a set of numbers. Coverage criteria, usually boolean conditions, are used to steer and stop the test generation process [6]. They are widely accepted means of assessing the quality of a test [7]. Both concepts will be taken up again later on in Section 4, where we discuss the coverage criteria of the technique that is

the basis of our proposal.

3 Previous Work

Usability testing of visualization is a well-studied area within visualization science. Related works, such as [8], [9], and [10], emphasize the need to assess whether a visualization is useful for its intended purposes. Without a doubt, we agree with this position, but we also emphasize that usability tests are only a part, a subset, of the types of tests that we must apply to a visualization. Usability tests do not evaluate functionality, which is the focus of this research. A peculiarity that emerges from the literature review is the number of visualization articles where the terms "testing", "verification", and "validation" were all used as synonyms for usability evaluations. Thus, this leads us to believe that there may be a lack of knowledge of the software V&V terminology within the visualization community.

Banerjee et al. [5] 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"). From the user's point of view, GUIs offer many degrees of usage freedom, i.e., users may choose to perform a given task by inputting GUI events in many different ways in terms of their type, number, and execution order. Banerjee et al. also provide a study of the existing body of knowledge [5] on GUI testing since 1991 and present a classification based on model-based GUI test techniques [11], as also did Memon and Nguyen [12]. Hellman et al. [13] presented a review of test-driven development of GUI. They stated that GUI testing is very difficult, in part, due to the degree of freedom GUIs allow users. GUIs can enter a large number of possible states in response to user input, and it is often difficult to determine the validity of a given state in an automated fashion. It becomes even more complex when we consider an information visualization technique.

Kazmi et al. [14] 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 should be; however, the authors do not present a system for this purpose. Although the model validity is not disputed, it is not possible to validate it 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. [15] 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. These include how data is obtained, represented, and interacted with. Although it is a broader vision than our proposal, the authors test the visualizations considering them as black boxes. Unlike the work of Kazmi et al., this one presents a case study on a particular map; however, the publication does not contain the set of rules used or how the semantic reasoner was used. Nor can it be understood from this test case how users' interactions affect the testing process.

Kirby and Silva [16] highlight the need to introduce verification and validation processes to the visualization development and the lack of research in this field within the visualization area. Larrea [17] also validates this last statement.

When considering these three types of testing mentioned, usability, GUI, and visualizations, there is a need to establish the difference between them. Or at least highlight why each of them cannot supplant the others. First, we must separate the usability tests from the other two. As Lauesen [18] said "Usability testing does not test the correctness of the program, but whether the user can work correctly and conveniently with it". Both GUI and visualization testing focuses on the functionality of the system, its correctness, and not its usability. Regarding GUI and visualization testing, as indicated by Banerjee et. al [5], GUI Testing deals with exercising the GUI's widgets (e.g., text boxes and clickable buttons). This 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 [19], is described in terms of the abstract data it represents. Information visualization is a more abstract visual representation than GUIs and therefore requires specific techniques for testing.

4 Black-box testing technique for information visualization. Sequencing constraints with low-level interactions

In 2017, we presented a new methodology [17] aimed at visualization testing through user interactions and from a black-box perspective. On that occasion, the technique was introduced without a supporting software tool. In this section, we briefly describe the methodology; refer to the original article for more information.

Our proposal introduced the concept of Sequence Constraint on the Interactions (SCI). Each SCI involves a set of binary or unary operators and a set of symbols. These symbols represent the actual interactions available in the visualization. In essence, each SCI is a regular expression formed by the interactions of the visualization that indicate the correct use of the visualization itself.

4.1 Sequence Constraint on the Interactions

Following the work by Kirani and Tsai [20], the operators involved in an SCI are:

Sequential: If an interaction I_2 must always go after the interaction I_1, then there is a sequential relationship among them, denoted as $I_1 \bullet I_2$.

Optional: If the user can choose between two interactions, said I_2 and I_1, then there is an optional relationship among them, denoted as $I_1|I_2$. Note that in this case, the notation $I_2|I_1$ is equivalent.

Repetition: If the user can use interaction I_1 multiple times in a row, then it is a repetition. Unlike the work done at [20] and [21], we introduce two types of repetition, one that implies that at least one time the user must use I_1 and the other that allows for zero appearance of I_1. The symbol $*$ represents cardinality 0 or more, and the symbol $+$, 1 or more. If I_1 can be used zero or more times, then this is represented as ${I_1}^*$. If I_1 must be used at least one time, it is expressed as ${I_1}^+$, which is equivalent to $I_1 \bullet {I_1}^*$.

These elements can be combined to form more complex expressions. If the user can use one of three interactions multiple times, this can be expressed as $(I_1|I_2|I_3)^+$. In this case, symbol $+$ indicates that at least one of the interactions must be used once. Repetition operators have precedence over Sequential and Optional operators. The Optional operator takes precedence over the Sequential one. Parentheses can be used to define the interpretation of an SCI. Suppose we have three interactions I_1, I_2, and I_3, then the following SCI

$${I_1}^+ \bullet I_2|I_3$$

expresses that first, we must consider the Sequential operator, *use* I_1 *one or more times*, and then we must choose between using I_2 or I_3. By using parentheses, we can change the interpretation of the SCI

$$({I_1}^+ \bullet I_2)|I_3.$$

In this case, we first consider the Optional operator to choose between using I_3 or the expression between the parentheses.

Let us imagine a visualization $\mathscr{V}$ with six interactions, *Open*, *Pan*, *Zoom*, *Selection*, *Detail*, and *Close*. *Open* represents the creation of the visualization, from opening the source data to setting up the visualization process; when *Open* concludes, the user has the actual visualization on screen. *Detail* represents detail on demand and can only be used if the user has previously selected something using *Selection*. *Pan* and *Zoom* allow the user to explore the visualization. The following grammar represents the constraints over the sequence of interaction in $\mathscr{V}$; notation considers O for *Open*, P for *Pan*, Z for *Zoom*, S for *Selection*, D for *Detail*, and finally, C for *Close*:

$$\text{SCI for } \mathscr{V} : O \bullet (O|Z|P|(S^+ \bullet D^*))^* \bullet C$$

This grammar states that the first valid interaction with $\mathscr{V}$ is *Open*, then the user can *Open* again, or *Zoom* or *Pan* or perform *Selection*. Note that if the user wants *Detail*, first, the user must complete at least one *Selection*. The interaction with $\mathscr{V}$ finishes when the user ends the visualization with the *Close* interaction.

4.2 Coverage Criteria

Within the presented proposal, two types of test cases are described: Valid test cases based on valid interaction sequences and invalid test cases based on interaction sequences that cannot be derived from the SCI.

Let I be the set of interactions available on the visualization $\mathscr{V}$, and G, the SCI for $\mathscr{V}$ 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 [21]: coverage criteria for valid sequences and invalid ones. The criteria for each category were defined for our technique.

4.2.1 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 sequences derived from G of length i. 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 sequences derived from G of length $i+1$.

Base+n Coverage: This is a generalization of the previous coverage. 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 sequences derived from G of length $i+n$, where $n \geq 2$. It is important to note that G may impose limits on how large n can be.

4.2.2 Coverage Criteria for Invalid Sequences

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

Invalid-2 Coverage: T satisfies the Invalid-2 Coverage Criteria if and only if T contains all the possible sequences obtained combining 2 interactions of I but are **not** derived from G.

Invalid-n Coverage: T satisfies the Invalid-n Coverage Criteria if and only if T contains all the possible sequences obtained combining n interactions of I, where $n \geq 2$, but are **not** derived from G.

4.3 Test Cases

Two lists of test cases are generated from the coverage criteria: a list of valid test cases and a list of invalid ones. Test cases generally have inputs, pre and post-conditions, and an expected result. The input for each test case is the subset of interaction that composes the case. Pre and post-conditions can be defined depending on the internal state of the system. Since the current grammar is not expressive enough to include conditions, this will be addressed in future works. Besides the result of the actual sequence of interactions, each test-case type has an expected result. Valid test cases are expected to run successfully, while it is expected that, at some point, the application will not allow invalid test cases to finish executing.

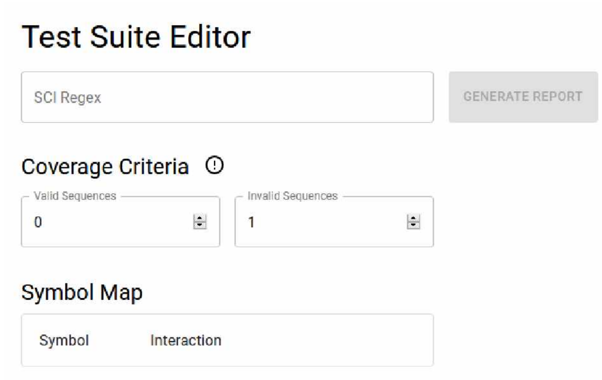
5 Our Proposal

The work developed in [17] does not include any support software tools. In this way, what was published served as a procedural manual. Our proposal in this work is to expand the development carried out by presenting the Test Suite Editor, a platform that implements the technique previously presented. The ultimate objective of the Test Suite Editor is automatically testing based on a given SCI. The platform, for the moment, is limited to generating the test cases, that is, sequences of interactions for selected coverage criteria. The execution of each test case must be carried out by one person. Each test case is presented in the form of documentation that the user can use to document the results of each run. To encapsulate the necessary SCI-parsing logic and to avoid coupling it with the rest of the application code, the project was subdivided into two parts that we will call SCI-Parser and SCI-App. The Test Suite Editor is currently available[1] and can be accessed from all standard web browsers. According to the concepts introduced in Section 2, this new tool is a black-box testing tool.

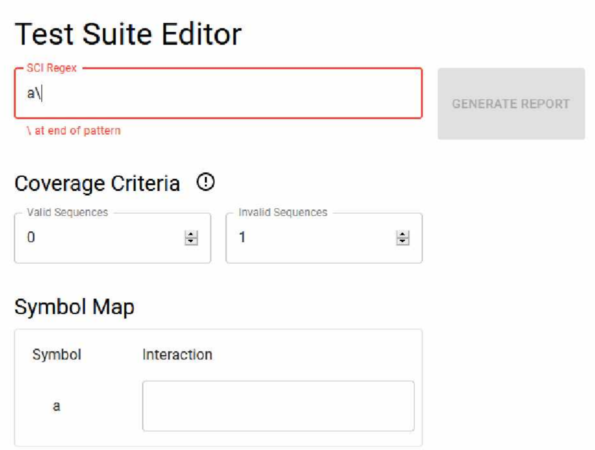
5.1 SCI-Parser

This module is in charge of validating potential SCIs and generating the expected test cases, according to the coverage parameters provided.

[1]https://cs.uns.edu.ar/~dku/vis/visualization-sci-testing/



(a) Home screen of the Test Suite Editor. It was developed using React as the front-end framework in TypeScript.



(b) If there is an error in the SCI, it is detected and reported by the application. The user's attention is obtained by employing a red box on the SCI input field and the indication of what the error was, which is located below the text field.

Figure 1: Test Suite Editor, a web platform that assists in visualization testing through user interactions.

SCI-Parser offers a small and simple API, composed of three static methods, briefly explained next. *isValid* receives as a parameter a text string that represents an SCI and returns a Boolean depending on whether it is a valid SCI or not. *syntaxErrorMessage* helps understand why a string is not a valid SCI. In case the SCI is valid, it returns null; otherwise, it returns a message that describes why the string is not a valid SCI. *parse* returns an instance of the SCI class as long as the string is valid; otherwise, it returns null.

The following are the elements available for the SCI class. The *interactionSymbols* attribute represents the set of symbols in the SCI, i.e. the user's interactions in the SCI. The *validSequences* method receives an optional parameter n (by default, it is zero) and returns the set of all valid sequences derived from the SCI that satisfies the *base+n* coverage criteria. *invalidSequences* is a method that receives an optional parameter n (by default, it is one) and returns the set of all invalid sequences that satisfies the *invalid-n* coverage criteria. In this way, the sequences of interactions,

the test cases, are generated. The generation of the interaction sequences is carried out using a third-party library (see 5.3.1).

5.2 SCI-App

All the development of the front-end was made following the usability principles presented in [22] and although it is still necessary to carry out formal usability tests following rigorous controls, the use that our peers have given the page allows us to affirm with confidence that it is easy to use.

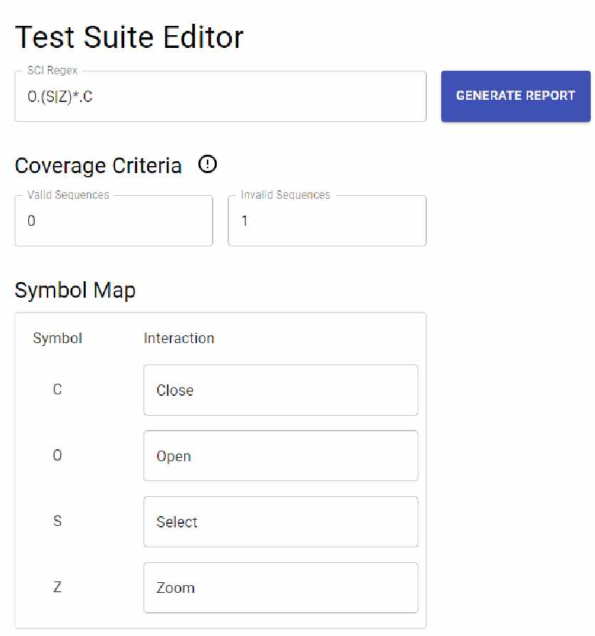
The front-end is divided into two parts. The initial one offers an editor that allows the user to enter the SCI, the coverage parameters, and an optional mapping between symbols and interaction names. Once the values are entered and validated, the user can access the second part of the application. This part presents a report with all the generated test cases, which can be completed as a form or exported as a PDF file.

The application offers an extremely simple editor. Figure 1(a) shows a screenshot in its initial state. As seen in the figure, it is composed of 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 in which the user can optionally enter the full name of the interaction, as shown in Figure 2(a).

Symbol mapping is optional at the individual level. It allows adding a more descriptive name for those symbols where is worthwhile and omitting it for those that the user considers unnecessary. When viewing the report, the names in the mapping are used to display a detailed version of the SCI expression and to list the interaction names in each test case. It is important to note that although the mapping is optional, it improves the readability of the generated report whatever abbreviated symbols are used. In case an error is detected in the entered values, a descriptive message is displayed (Figure 1(b)). These error messages are provided by the SCI-Parser module. Once the required values are entered, the user is enabled to generate the report. The newly generated report is displayed in a new tab (Figure 2(b)).

The report was designed and implemented with simplicity in mind so that the same format presented in the application could be exported as PDF using the standard printing functionality directly. At the top, it has a heading that shows the values previously entered in the editor, from which the test cases report was generated. The blue PDF icon is the button that allows users to export the document as a PDF file (Figure 3).

Then the corresponding test cases are listed, which are grouped according to whether they are valid or



(a) Each SCI is written using letters as regular expression symbols. However, it is possible to map these symbols to strings for ease of understanding. This information can be entered in the Symbol Map table. This step is optional, and it is the application that is in charge of automatically detecting the symbols used and enabling a text field for each one in this table. When a symbol has a defined mapping, that character string is used in the test case report.

(b) Once the SCI is validated, it is possible to generate the report of test cases. It is generated from the indicated coverage criteria. This report can be used as a web form or as a PDF. The report allows indicating if each interaction could be carried out, as well as the complete sequence. There is also space to enter comments related to each sequence.

Figure 2: Test Suite Editor allows the generation of reports that help in the execution of test cases.

11/17/2020 SCI App

Report

SCI: 0.(S|Z)*.C
Mapped SCI: Open.(Select|Zoom)*.Close
Coverage criteria: Base + 3 Invalid + 3

Test cases for valid sequences

Valid test case 0.C
- 0. Open
- 1. Close

Comments

Valid test case 0.S.C
- 0. Open
- 1. Select
- 2. Close

Comments

Valid test case 0.Z.C
- 0. Open
- 1. Zoom
- 2. Close

Comments

Valid test case 0.S.S.C
- 0. Open
- 1. Select
- 2. Select
- 3. Close

Comments

Valid test case 0.S.Z.C
- 0. Open
- 1. Select
- 2. Zoom
- 3. Close

Comments

Valid test case 0.Z.S.C
- 0. Open
- 1. Zoom
- 2. Select
- 3. Close

Comments

localhost:3000/report/099f0a15-45f5-48be-a3d6-01601d87ccdf 1/14

Figure 3: The same report used as a form can be exported as a PDF, keeping the same presentation.

invalid sequences of interactions. Each test case is represented by a box like the one shown in Figure 4, where the title shows the sequence of interactions from the SCI expression that results in the test case. As mentioned above, in cases where mapping was provided, the names will be used to list interactions. The report allows users to check whether the test case was successfully executed or failed at some point. After the execution of the test case, 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.

5.3 Technical notes

5.3.1 SCI-Parser

The SCI-Parser code is available in its respective repository on GitHub[2]. As can be seen there, this module was not implemented from scratch but was started from a fork of the genex.js project repository [23], authored by Alix Axel. SCI-Parser was implemented entirely in TypeScript, to make it easier to use by providing a statically typed API. Like genex.js, SCI-parser uses the `ret` (Regular Expression Tokenizer) library [24]

[2] `https://github.com/mschiaffino/sci-parser`

Figure 4: These are the fields with which the user can interact in the report.

to parse the regular expression associated with the SCI and return a tree of tokens. This tree is then traversed to generate the strings that represent the test cases.

5.3.2 SCI-App

The application was developed using React as the front-end framework; it was programmed in TypeScript to be consistent with the SCI-Parser module and to take advantage of static typing. In addition, the Material-UI web component library [25] was used. As its name indicates, this library offers a wide variety of components developed following the Material Design standards [26].

5.3.3 Limitations

The Test Suite Editor presents two technical limitations. Both are a direct consequence of the comprehensiveness of test cases generated by the testing technique. When the coverage criteria imply the generation of a huge amount of interaction sequences for the given grammar, displaying the serialized report becomes problematic for the browser to handle.

The first limitation manifests when the space required to store the serialized report is larger than the available browser's local storage. Since the serialized report cannot be stored, it is impossible to open the new tab to display it. The second limitation occurs when the size of the serialized report is not large enough to exhibit the first limitation, but the number of React components to render becomes unmanageable for the browser. In this case, after a while, the tab that should display the report is aborted by the browser. A potential solution is to implement some virtualization to avoid rendering not visible components. However, due to the variable size of the test case components, this would not be trivial to implement. In this instance, seeking a solution to these limitations was not a priority; both are present for combinations of grammar and criteria where it would be impractical for a person to verify all the generated test cases.

6 Test Cases

In this section, we show how the Test Suite Editor detected possible errors in two different interactive web visualizations. The notation for the sets of valid and invalid sequences will be as follows:

- T_{+n} denotes a set of sequences that satisfy the base$+n$ coverage criteria. Thus, all the sequences in this set are valid sequences of the minimum possible length plus n.

- T_n^{inv} denotes a set of sequences that satisfy the invalid-n coverage criteria. Thus, all sequences in this set are invalid and of length n.

6.1 MoCap Synchromparator

MoCap Synchromparator [27] is a comparative visualization of motion capture sequences that focuses on the time dimension. The visualization starts with an overview of the misalignment between the data corresponding to different subjects (Figure 5). Details on the comparison between two particular sequences can be obtained on demand by clicking on each overview box (Figure 6). The detail view provides an overview of the misalignment between the selected sequences and visual information about when one of them is delayed or early with respect to the other. The time frames where the sequences differ are easily perceptible due to color-coding.

The visualization offers four interactions: Open, which loads the data sets and creates the visualization; Hover, which displays a brief information text every time the user places the mouse over one of the boxes from the overview; Click, after the hover interaction, the user can click in the box and display the detail view; and Back, on the detail view, the only available interaction is to return to the overview, this interaction is achieved by clicking the blue arrow on the bottom right. Once the visualization is created, the user can hover as many times as wanted, click to get more details, and then back to the overview to continue exploring the visualization. The following SCI represents this interaction; it uses the symbols O for Open, H for Hover, C for Click, and finally B for Back:

$$O^+ \bullet H^* \bullet (H^+ \bullet C \bullet B)^*$$

From this SCI, the online tool generated both valid and invalid interaction sequences. The minimum possible values for the valid and invalid coverage criteria are 0 and 1, respectively, which implies sequences of length 1:

$$T_{+0} = \{O\}$$
$$T_1^{\text{inv}} = \{H, C, B\}$$

The only valid sequence is the Open interaction, which worked correctly. The set of 1-length invalid sequences T_1^{inv} includes all the other interactions: Hover, Click, and Back. None of those sequences could be executed, which is correct. Increasing the coverage criteria by 1 resulted in the following T_{+1} and T_2^{inv} sets of sequences that satisfy the base$+1$ coverage criteria and the invalid-2 coverage criteria, respectively:

$$T_{+1} = \{O \bullet O, O \bullet H\}$$
$$T_2^{\text{inv}} = \{B \bullet B, B \bullet C, B \bullet H, B \bullet O, C \bullet B, C \bullet C, C \bullet H, C \bullet O, H \bullet B, H \bullet C, H \bullet H, H \bullet O, O \bullet B, O \bullet C\}$$

The only two valid sequences were executed flawlessly. However, from the set of invalid sequences of length 2, we found a problem with sequence $O \bullet C$, which starts with the valid interaction Open but continues with an invalid one. Surprisingly, it was possible to execute this sequence. Let's not forget that this occurs in the context of invalid sequences, which means that being able to execute one of these sequences indicates the presence of an error. When the user clicks outside but close to an overview box (Figure 7), the system considers it as a click on the box and moves to the detailed view. Even in the case depicted in Figure 7, where the click is closer to the overview of *trial0005* × *trial0008*, the system recognizes it as a click on *trial0005* × *trial0005*. No other errors were found after this one. Figure 8 shows the report of test cases. Note that all three valid cases were executed satisfactorily. However, it was possible to execute completely an invalid case, then that was an unsatisfactory invalid case.

6.2 Spinel Web

The second application is Spinel Web [28], an interactive web application for visualizing the chemical composition of spinel group minerals. The spinel group minerals provide useful information regarding the geological environment in which the host rocks were formed. These minerals constitute excellent petrogenetic indicators and guide the search for mineral deposits of economic interest. It is common to represent the spinels' mineral composition in a prismatic space called spinel prism.

The Spinel Web provides a rich set of functionalities required by the geologist, comprising 2D binary plots, ternary plots, and a 3D representation of the spinel prisms. All views are interactive, linked, and integrated into a coordinated multiple views setup, allowing the dataset to be simultaneously displayed using different visualization techniques. The overall premise of this exploratory technique is that users better understand their data if they can interact with their data by viewing it through different representations.

A common task in spinel mineral analysis is to explore the data in the spinel prism context, analyzing the representation of the dataset in the spinel prism and its projections simultaneously (Figure 9).

In order to illustrate the usefulness of Test Suite Editor in the detection of errors, we consider a partial evaluation focused on the visualization of one dataset in the spinel prism context, considering only two coordinated views to perform the testing: the spinel prism

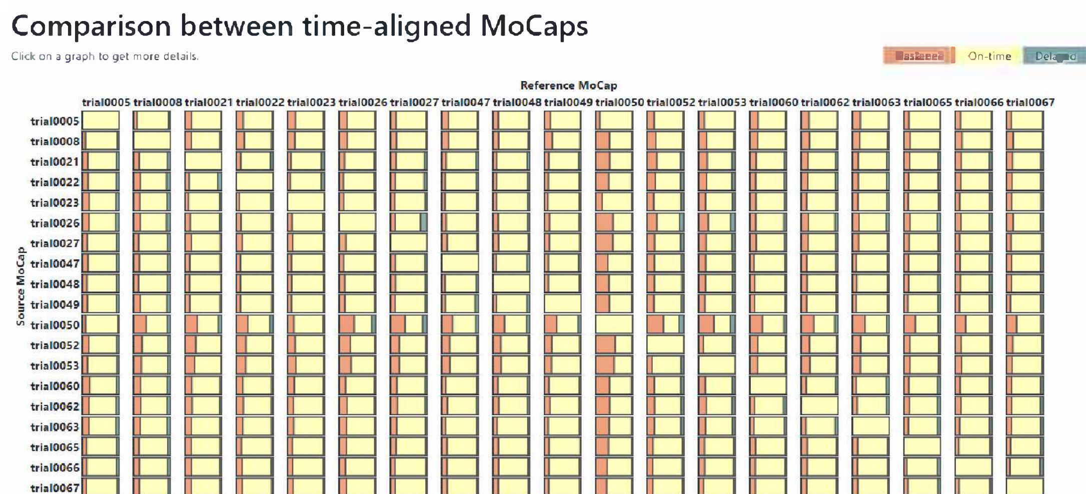


Figure 5: The visualization starts with an overview of the misalignment between the data corresponding to different subjects. The visual representation is created as a matrix of the percentage summaries of every pair of sequences.

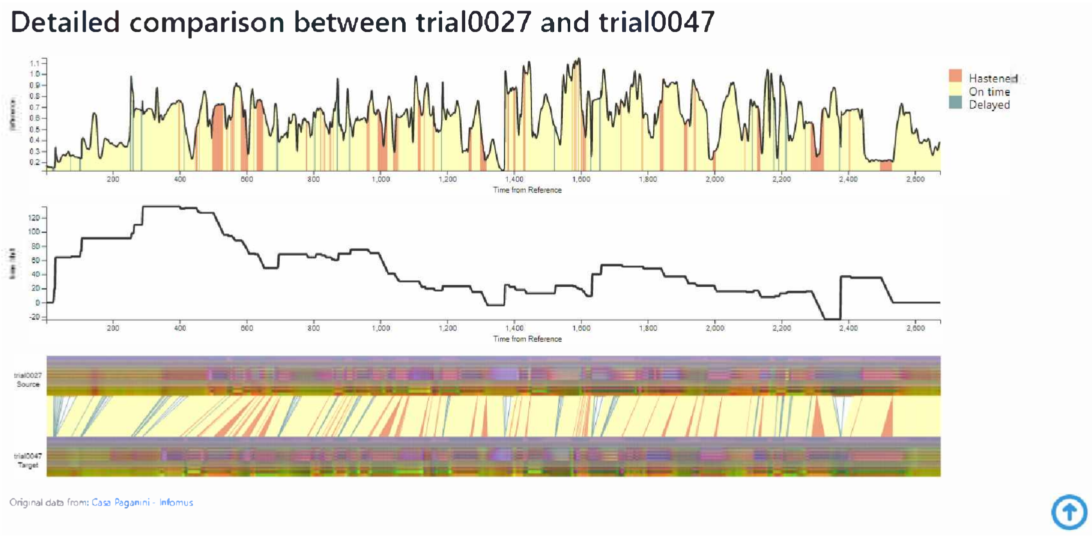


Figure 6: Details on the comparison between two particular sequences. The upper graph shows, with colors, the misalignment between the motion captures and the absolute difference between time-aligned frames. The middle graph is the misalignment function. The graph at the bottom shows a heat-map visualization of the two motion captures and how they are aligned in time. The color coding uses blue for delayed frames, yellow for on-time ones, and red for early ones.

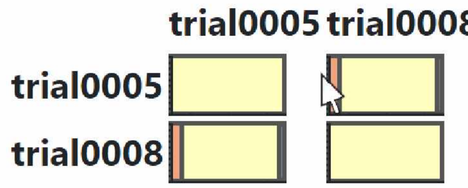


Figure 7: The system considers some clicks outside an overview box as actual clicks on a box, and the system goes to the detailed view.

and the triangular projection. For this partial evaluation, we consider four interactions: open the data file, load the spinel prism, load the triangular projection, and brushing and linking. The brushing-and-linking interaction allows the user to interactively select (brush) subsets of the data in a view, and all the corresponding data items in all linked views will be consistently highlighted (linking).

To write the Sequencing Constraints with Low-Level Interactions we replace each interaction with a simplified representation: let O be open the data file, P to load spinel prism, T to load triangular projection, and B to do brushing and linking. To visualize the data, the first interaction that must explicitly occur is to open the data file. Then one or both charts must be loaded, but there is no restriction in the order on which of the two low-level interactions P and T should be used. The only restriction is that the same chart must not be loaded more than once, but it is not necessary to load both for the application to work properly. The B interaction can be used as soon as the first chart is loaded. Therefore, the behavior of this visualization is described by the following SCI:

$$O|(O\bullet((P\bullet B^*)|(P\bullet B^*\bullet T)|(T\bullet B^*)|(T\bullet B^*\bullet P))\bullet B^*)$$

Since it is mandatory to open the data file as the first action before loading a chart, the minimum sequence of interactions valid for this visualization is 1. The test set T_{+0} that satisfies the Base Coverage criteria contains only one interaction, the Open the data file interaction, that worked properly:

$$T_{+0} = \{O\}$$

With the Test Suite Editor, we easily generated the test sets that satisfy the Base+1, Base+2, and Base+3 Coverage criteria:

$$\begin{aligned}
T_{+1} &= \{O\bullet P, O\bullet T\} \\
T_{+2} &= \{O\bullet P\bullet B, O\bullet P\bullet T, O\bullet T\bullet B, O\bullet T\bullet P\} \\
T_{+3} &= \{O\bullet P\bullet B\bullet B, O\bullet P\bullet B\bullet T, O\bullet P\bullet T\bullet B, \\
&\quad O\bullet T\bullet B\bullet B, O\bullet T\bullet B\bullet P, O\bullet T\bullet P\bullet B\}.
\end{aligned}$$

Figure 10 shows the report generated by the Test Suite Editor for the valid sequences that satisfy the Base+3 Coverage criteria. This report was very useful to guide in the testing of the Spinel Web, and fortunately, no errors were found when executing the interaction sequences in T_{+1}, T_{+2}, and T_{+3}.

We also used the Test Suite Editor to generate the test cases for invalid sequences of interactions (see Figure 12). The test set T_1^{inv}, which contains all the possible sequences of length 1 not derived from the SCI, satisfies the invalid coverage criteria. In this particular case, it includes any interaction other than open the data file.

$$T_1^{\text{inv}} = \{P, B, T\}$$

The Spinel Web worked properly, not allowing to load any view or perform a brush before opening a data file.

Then, we generated the test sets T_2^{inv} and T_3^{inv} containing the invalid sequences that satisfy the Invalid-2 and Invalid-3 Coverage.

$$\begin{aligned}
T_2^{\text{inv}} = \{&B\bullet B, B\bullet O, B\bullet P, B\bullet T, O\bullet B, O\bullet O, P\bullet B, \\
&P\bullet O, P\bullet P, P\bullet T, T\bullet B, T\bullet O, T\bullet P, T\bullet T\}
\end{aligned}$$

$$\begin{aligned}
T_3^{\text{inv}} = \{&B\bullet B\bullet B, B\bullet B\bullet O, B\bullet B\bullet P, B\bullet B\bullet T, \\
&B\bullet O\bullet B, B\bullet O\bullet O, B\bullet O\bullet P, B\bullet O\bullet T, \\
&B\bullet P\bullet B, B\bullet P\bullet O, B\bullet P\bullet P, B\bullet P\bullet T, \\
&B\bullet T\bullet B, B\bullet T\bullet O, B\bullet T\bullet P, B\bullet T\bullet T, \\
&O\bullet B\bullet B, O\bullet B\bullet O, O\bullet B\bullet P, O\bullet B\bullet T, \\
&O\bullet O\bullet B, O\bullet O\bullet O, O\bullet O\bullet P, O\bullet O\bullet T, \\
&O\bullet P\bullet O, O\bullet P\bullet P, O\bullet T\bullet O, O\bullet T\bullet T, \\
&P\bullet B\bullet B, P\bullet B\bullet O, P\bullet B\bullet P, P\bullet B\bullet T, \\
&P\bullet O\bullet B, P\bullet O\bullet O, P\bullet O\bullet P, P\bullet O\bullet T, \\
&P\bullet P\bullet B, P\bullet P\bullet O, P\bullet P\bullet P, P\bullet P\bullet T, \\
&P\bullet T\bullet B, P\bullet T\bullet O, P\bullet T\bullet P, P\bullet T\bullet T, \\
&T\bullet B\bullet B, T\bullet B\bullet O, T\bullet B\bullet P, T\bullet B\bullet T, \\
&T\bullet O\bullet B, T\bullet O\bullet O, T\bullet O\bullet P, T\bullet O\bullet T, \\
&T\bullet P\bullet B, T\bullet P\bullet O, T\bullet P\bullet P, T\bullet P\bullet T, \\
&T\bullet T\bullet B, T\bullet T\bullet O, T\bullet T\bullet P, T\bullet T\bullet T\}
\end{aligned}$$

We used the generated report (see Figure 12) to verify all the invalid sequences. Unexpectedly, Spinel Web did not work properly with all test cases of T_2^{inv}. It was possible to run without a problem the invalid sequence $O\bullet O$, evidencing that the system allows opening data files more than once. At this point, we realized that this problem was going to persist while testing the Spinel Web with the invalid sequences of T_3^{inv}. Hence, it was not a surprise that the system allowed to perform the invalid sequences $\{O\bullet O\bullet O, O\bullet O\bullet P, O\bullet O\bullet T, O\bullet P\bullet O, O\bullet T\bullet O\}$, which involve multiple data opening. However, we did not expect the Spinel Web to allow the execution of the invalid sequences $\{O\bullet T\bullet T, O\bullet P\bullet P\}$, revealing a second error: the system allows loading the same view more than once (Figure 11). Finally, thanks to the automatic generation of cases and the interactive

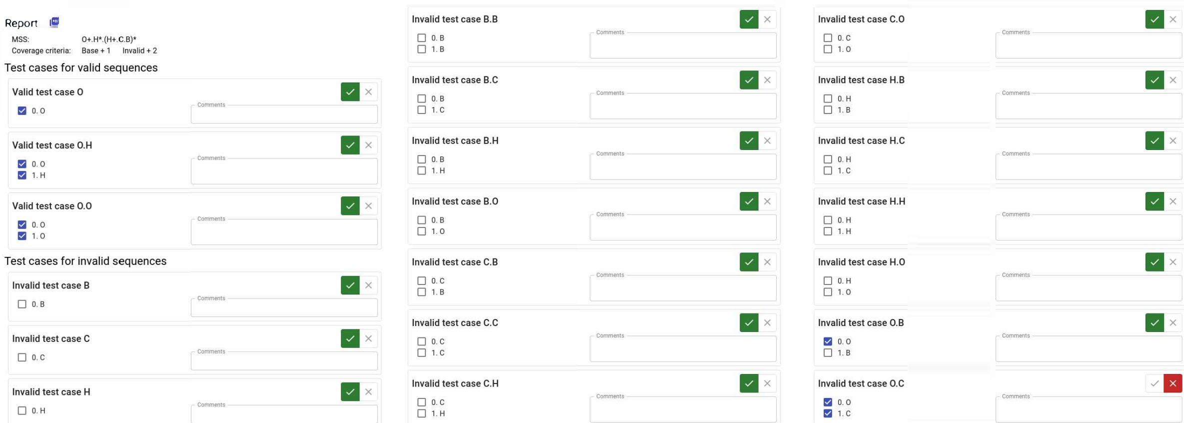


Figure 8: Report of test cases with coverage criteria Base+0 and Base+1 for valid sequences and invalid-1 and invalid-2 for invalid ones. Note that it was possible to execute the last invalid case, then it is an unsatisfactory test case.

report provided by the Test Suite Editor, we were able to detect two important errors in the Spinel Web that were definitely overlooked in previous stages of development.

7 Conclusions and future work

Today's decision-making processes require the assistance of specialized tools, which include information visualizations and interactions. The responsibility of developers of these tools is to ensure their correct operation, especially when the impact of decisions based on the displayed information is critical to human life. This crucial usage motivated us to develop a new tool that assists in visualizations testing and ensures their correct operation. This new tool is easy to use and does not require specific knowledge in the V&V area; it is platform-, implementation-, and language-independent, it applies to any visual representation. As we mentioned before, it is not our goal to provide a contribution to the testing community about information visualization, but instead, our contribution is to the information visualization community about testing. As detailed case studies demonstrated the tool allows the users to find errors that would not otherwise be easy to detect. During the tests documented in this work, the platform did not exhibit any problem related to the technical limitations previously described. The cases tested were representative combinations of visualization use and it was considered that it was not necessary to continue with greater coverage criteria. However, the coverage criteria continued to be increased to establish at what point our proposal began to exhibit problems related to technical limitations. For example, in the MoCap Synchromparator it was only when the base+4 and invalid−5 criteria were met that the platform experienced issues and was unable to generate the full set of test cases.

There is still work to do; specifically, we are looking to generate a more expressive grammar to represent new conditions in the interaction sequence. But above all, we are looking to automate the execution of the test cases generated from a regular expression and the coverage criteria. Regarding the current web implementation, we will look for alternatives to overcome the browser's memory limitation and be able to display long reports. Furthermore, given the grammar, we intend to inform the user of maximum values for the coverage criteria that reach a manageable number of test cases.

Competing interests

The authors have declared that no competing interests exist.

Funding

This work was partially supported by the following research projects: PGI 25/N050 and PGI 24/N048 from the Secretaría General de Ciencia y Tecnología, Universidad Nacional del Sur, Argentina, PICT-2017-1246 by ANPCyT (Argentina), and PDTS-0414 by ANPCyT (Argentina) and Universidad Nacional del Sur (UNS).

Authors' contribution

MLL and DKU carried out the conception of this work. MS carried out the implementation of the web platform. MLL performed the state of the art, DKU and MLG worked on the test cases. MLL, DKU and MLG worked on the general writing of the article and the reviews provided.

Figure 9: Screenshot of an analysis session in which geologists interact with different views of the data: the spinel prism (left), and the triangular (bottom right) and lateral projection (upper right). A brushing is activated (red points) and highlighted in all coordinated views [28].

References

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

[2] M. Nouman, U. Pervez, O. Hasan, and K. Saghar, "Software testing: A survey and tutorial on white and black-box testing of c/c++ programs," in *2016 ieee region 10 symposium (tensymp)*, pp. 225–230, IEEE, 2016.

[3] 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.

[4] M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, and S. Shaikh, "Formal methods for software engineering: Languages, methods, application domains," 2020.

[5] 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.

[6] S. Sherin, M. Z. Iqbal, M. U. Khan, and A. A. Jilani, "Comparing coverage criteria for dynamic web application: An empirical evaluation," *Computer Standards & Interfaces*, vol. 73, p. 103467, 2021.

[7] M. Friske, B.-H. Schlingloff, and S. Weißleder, "Composition of model-based test coverage criteria.," in *MBEES*, pp. 87–94, 2008.

[8] Z. Štěrba, Č. Šašinka, Z. Stachoň, *et al.*, "Usability testing of cartographic visualizations: principles and research methods," in *Proceedings of the 5th International Conference on Cartography and GIS Proceedings*, vol. 1, pp. 147–256, 2014.

[9] Á. Vizoso, "Information visualization and usability: Tools for human comprehension," in *Journalistic Metamorphosis*, pp. 85–98, Springer, 2020.

[10] 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.

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

[12] 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* (M. V. Zelkowitz, ed.),

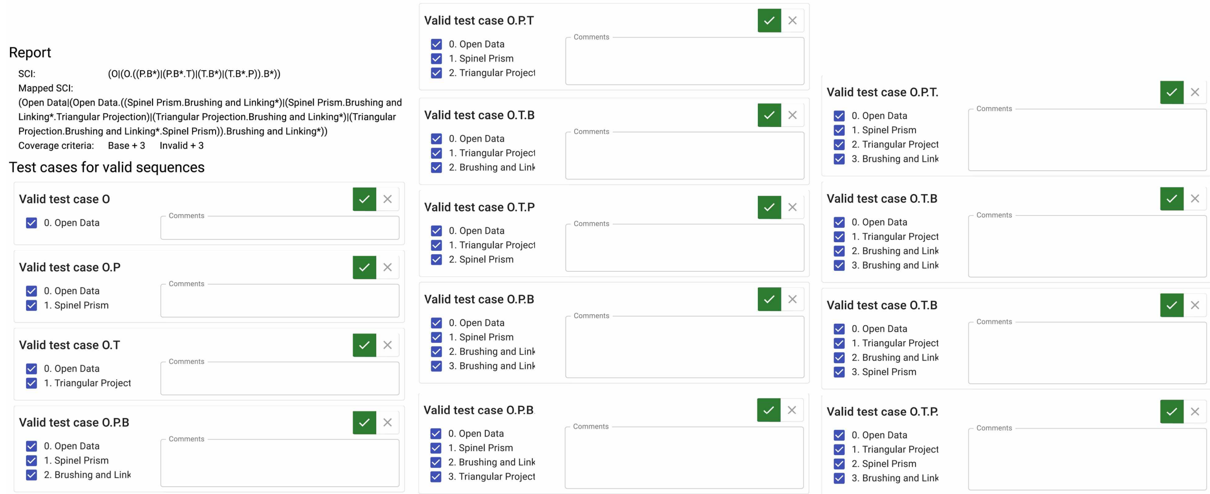


Figure 10: The report generated by the Test Suite Editor for the valid sequences that satisfy the Base+3 Coverage criteria. Fortunately Spinel Web worked properly for all the test cases that satisfy the Base+3 Coverage criteria.

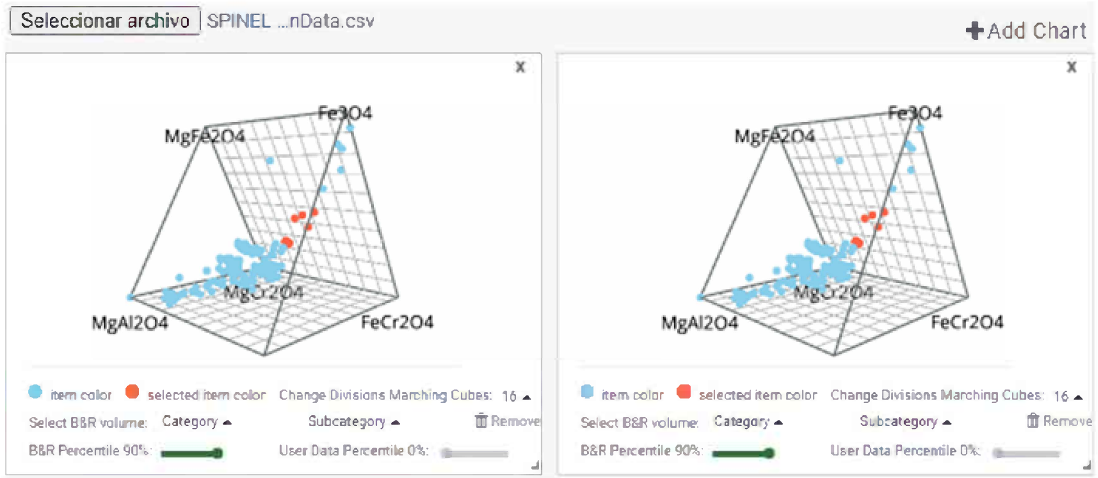


Figure 11: Screenshot of an analysis session in which the system allowed loading the spinel prism view twice.

vol. 80 of *Advances in Computers*, pp. 121–162, Elsevier, 2010.

[13] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer, *Agile Interaction Design and Test-Driven Development of User Interfaces – A Literature Review*, pp. 185–201. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[14] S. H. Kazmi, F. Azam, M. W. Anwar, and B. Maqbool, "A meta-model for automated black-box testing of visualization based software applications," in *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, pp. 183–187, 2020.

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

[16] 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.

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

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

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

[20] S. H. Kirani and W. Tsai, *Specification and verification of object-oriented programs*. PhD thesis, Citeseer, 1994.

[21] F. Daniels and K. Tai, "Measuring the effectiveness of method test sequences derived from se-

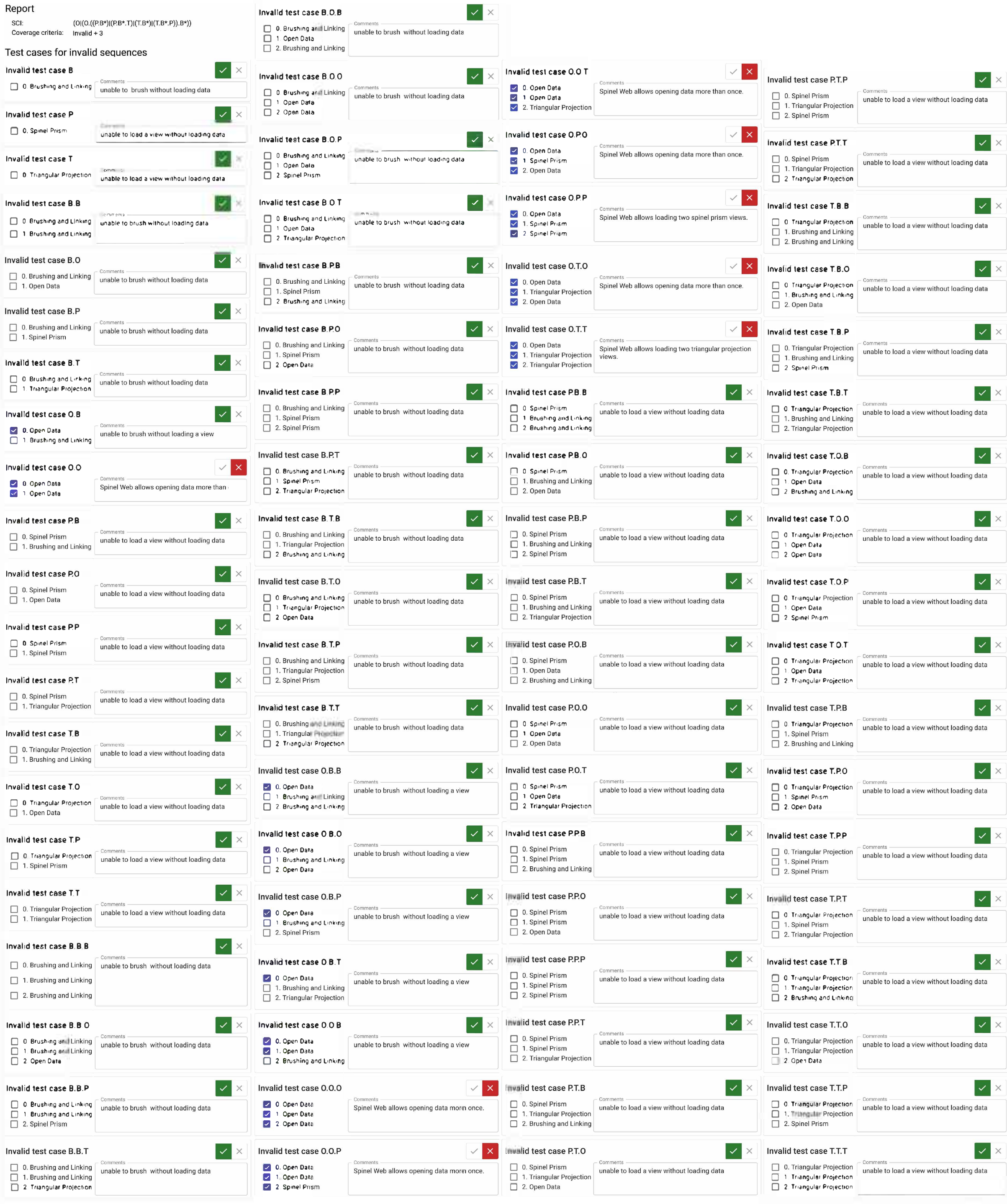


Figure 12: The report generated by the Test Suite Editor for the invalid sequences that satisfy the Invalid-2 and Invalid-3 Coverage. Unfortunately the Spinel Web did not work properly with all test cases.

quencing constraints," in *Proceedings of Technology of Object-Oriented Languages and Systems-TOOLS 30 (Cat. No. PR00278)*, pp. 74–83, IEEE, 1999.

[22] J. Nielsen, "Usability inspection methods," in *Conference Companion on Human Factors in Computing Systems*, CHI '94, (New York, NY, USA), p. 413–414, Association for Computing Machinery, 1994.

[23] A. Alix, "genex.js project repository." `https://github.com/alixaxel/genex.js/`. Accessed: 2021-04-16.

[24] R. S. Engelschall, "Regular expression tokenizer library." `https://github.com/rse/tokenizr`. Accessed: 2021-04-16.

[25] Material-UI, "Material-ui. a popular react ui framework." https://material-ui.com/. Accessed: 2021-04-16.

[26] I. G. Clifton, *Android user interface design: Implementing material design for developers*. Addison-Wesley Professional, 2015.

[27] D. K. Urribarri, M. L. Larrea, S. M. Castro, and E. Puppo, "Overview+detail visual comparison of karate motion captures," in *Computer Science – CACIC 2019* (P. Pesado and M. Arroyo, eds.), (Cham), pp. 139–154, Springer International Publishing, 2020.

[28] A. S. Antonini, M. L. Ganuza, G. Ferracutti, M. F. Gargiulo, K. Matković, E. Gröller, E. A. Bjerg, and S. M. Castro, "Spinel web: an interactive web application for visualizing the chemical composition of spinel group minerals," *Earth Science Informatics*, vol. 14, no. 1, pp. 521–528, 2021.

Citation: 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
Received: April 27, 2021 **Accepted:** November 2, 2021.
Copyright: This article is distributed under the terms of the Creative Commons License CC-BY-NC.