Using Provengo for Bootstrapping Model-Based Testing (Experience Paper)

Yeshayahu Weiss, Gera Weiss, and Achiya Elyasaf,

***Abstract*—We share our experience with Provengo, a tool based on behavioral programming designed to address the challenges faced by users of model-based testing (MBT). MBT leverages a model of the required behavior of the system under test to automatically generate and evaluate test cases. However, constructing a model for a system with unclear requirements and updating it when new crosscutting requirements are introduced can be daunting. Provengo simplifies the process by allowing testers to write simple stories that describe individual aspects of the system behavior and combine them automatically. This approach caters to the incremental introduction of crosscutting requirements, and Provengo also supports abstractions and test coverage mechanisms to assist the bootstrapping and test generation process. We showcase our experience using Provengo to test the multi-session feature of the Magento-2 e-commerce platform, an example of testing a black-box system without a complete specification. We incrementally built a simple model using Provengo, which focuses on a specific aspect of the system's behavior and successfully detected several errors. We discuss how Provengo can effectively address the challenge of bootstrapping MBT and identify areas where the tool can be further enhanced.**

***Index Terms*—Model-Based Testing, Behavioral Programming**

I. Introduction

Model-based testing (MBT) [1], [2] is a technique for testing software systems that generates test cases from a model that captures the software's expected behavior. MBT has many benefits and enables testing systems that cannot be tested without a model when it is impossible to manually define a small set of representative scenarios and specify the expected system behavior for each run. However, MBT also has some drawbacks, such as creating and updating models, especially for complex systems and by testers not skilled in formal methods [3], [4]. Guidelines for MBT usually say that "it is best suited for the initial stage of the product, as things are still very small. It is easy to integrate with the system requirements then because as things get bigger, you get to update just the model" [5]. However, since testing often starts after the system is built and without full specification of the requirements, we explore the challenge of gradually applying model-based testing to an existing system.

In this paper, we share our experience with Provengo. This new tool addresses the challenge of bootstrapping MBT by enabling testers to describe natural user stories similar to standard non-model-based tests. With the addition of anti-scenario specifications, the tool can integrate these stories into a model that produces many executions traces through smart interleaving. Provengo supports quick and incremental model development by defining abstract user stories that generate concrete interactions with the system via high-level events.

To demonstrate the capabilities of Provengo, we present an experiment in which we rapidly tested a small part of a large system without a formal specification. In particular, we tested the multi-session behavior of the Magento e-commerce platform. We tested the system's behavior when users simultaneously purchase using multiple browser sessions. This experiment assessed how well Provengo can handle the complexity of testing a system without a complete specification and how it can help identify system behavior issues.

Testing an online store with multiple sessions reflects how real customers use e-commerce websites. Customers often open multiple tabs to compare products and prices. These actions create parallel sessions that interact with the same online store and affect its state and behavior. For example, purchasing items in one session affects their availability in another session, changing prices by the admin immediately affects users in other sessions, and so on. Testing the correct behavior in these situations requires generating and executing many test cases that cover different combinations of actions and outcomes across multiple sessions. However, traditional testing methods are not feasible or effective for this scenario. Manual testing, for example, is too slow and error-prone to cover all cases, and scripted testing is too rigid and brittle to handle multiple sessions' dynamic and complex nature. Therefore, we need a different approach to automate and optimize the test case generation and execution.

To explain the contribution of our paper, a search for the most common challenges presented by most MBT practitioners and the finding are:

1) It demands a high amount of investment as well as effort [6] [7] [8]
2) It requires skilled and disciplined software [9] testers [6] [8]
3) The first test case takes longer to generate as it requires more advanced work than traditional manual testing [10].
4) The learning curve of model-based testing is pretty steep, and its complexity makes it harder to understand for beginners. [11] [8]
5) The choosing an approach to support the MBT process due to the variety of models and tools availability [10]

In light of this finding our paper examines how Provengo's tools can improve MBT by addressing these challenges:

1) **RQ1:** Is it possible to quickly initiate an effective testing process using the scenario-based modeling approach

supported by the tool?
2) **RQ2:** Can modeling be done without requiring knowledge of formal methods?
3) **RQ3:**Is it feasible to apply model-based testing on an existing system without an explicit requirement document?
4) **RQ4:** Can the model be used to generate test suites that cover specific areas of interest?
5) **RQ5:** Is it possible to discover new bugs in a real system using this approach?

The paper is organized as follows: First, we provide an overview of the challenges associated with testing multi-session behavior and, more generally, detecting high-level race conditions in reactive systems, as well as the limitations of conventional testing approaches in this context. Then, we introduce the story-based approach and Provengo tool, a novel testing tool that enables systematic test-case generation and provides comprehensive coverage of the system behavior. We then detail our methodology using Provengo to analyze the Magento-2 e-commerce platform, showcasing its effectiveness through two identified bugs. In addition, we present an examination of test case selection, minimization, prioritization, and evaluation. We summarize the results and discuss the implications of this new test case generation method. To optimize readability and space utilization, we integrated the related work throughout the paper.

II. Provengo's Story-Based Testing Approach

Story-based testing (SBT) is a type of system and software testing that involves creating a narrative around users and their interactions with a system or application [12]. This narrative defines the test cases and expected outcomes of each test. SBT can help testers focus on the user perspective and the business value of the system or application rather than on the technical details. SBT can facilitate communication and collaboration among testers, developers, and stakeholders.

The Provengo tool uses SBT to enable testers to create models for MBT. Specifically, the tool allows testers to write natural user stories similar to standard non-model-based tests and combine them into a model that yields many execution traces by smart interleaving—a technique that creates different combinations of user actions and system responses by interleaving the user stories [13].

Using Provengo, testers can create models for MBT without learning formal methods or writing complex code. They can also test different system or application aspects by focusing on user stories and high-level events. Provengo can generate many test cases from user stories and execute them using Selenium [1], report on the results, and identify system errors.

A key challenge in SBT is considering all possible user interactions, especially in complex systems with many features and options. This is because the number of possible interactions can quickly become very large, making it difficult to anticipate and test all of them. Also, users may not be familiar with the full range of features and capabilities, making it challenging to develop comprehensive test cases.

This paper describes a solution that helps detect errors in reactive computer systems by automatically generating test cases. Specifically, we focus on a type of error known as a 'high-level race condition,' which can occur when multiple system parts interact unexpectedly. By automating the test case generation process, our approach ensures that all possible combinations of user interactions are explored, giving us a complete picture of how the system behaves. This saves time and effort compared to manual testing and also ensures that we can thoroughly examine the system for any potential issues [15].

Consider, for example, the following base story, in an e-commerce system: "An customer: (1) Logs in and navigates to the products page; (2) Adds several items to the cart; and (3) Checkouts."

From this base story, it is possible to generate several linear test cases that follow the same sequence of steps. For example, one test case might involve adding products under sale while another will add products with an age restriction.

However, it is not always easy for humans to think of non-linear combinations of stories, such as a customer that adds these items, while, concurrently, an admin changes its price. This scenario would be complex for humans to think of and manually test, though it could be easily generated by Provengo, which automatically interleaves the base stories.

Note that generating all possible ways to interleave the base stories is not always sufficient. This is because the complete set of generated test cases can be enormous and may include many cases that are unlikely to occur in practice or are irrelevant to the user's needs. Additionally, the complete set may include scenarios that are impossible or unrealistic, such as scenarios where actions are performed out of sequence or in a way that violates the rules or constraints of the system.

To address these issues, it is often necessary to impose constraints on the interleaving of base stories. For example, it may be necessary to require that a product must be visible to clients before they can add it to their cart. Constraints can help to ensure that the model and the generated test cases are relevant and realistic and can help to reduce the number of test cases that need to be generated and evaluated.

In this paper, we demonstrate how we use the Provengo tool for specifying stories and constraints, executing and analyzing them, and generating effective, yet small, test suites. The following section discusses some of the details and considerations for this approach.

III. The Provengo Tool

Provengo Technologies is a company that presents a next-generation approach to software and product development. By providing the Provengo system with simple instructions, users can automate and synthesize all interactions and communications related to product definition between all teams involved in the development process [16]. The system enables users

[1]Selenium [14] is an open-source automated testing framework widely used for web application testing. Enabling developers to write test scripts that simulate user interactions with web browsers and validate expected functionalities.

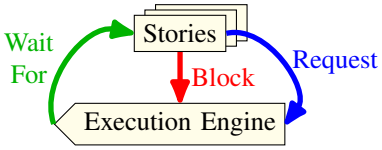

Fig. 1. An illustration of the execution cycle of a system where stories specify events they want to be triggered and events they want to block. The execution engine is responsible for selecting an event that was requested and not blocked and then notifying all the stories that waited for or requested that event.

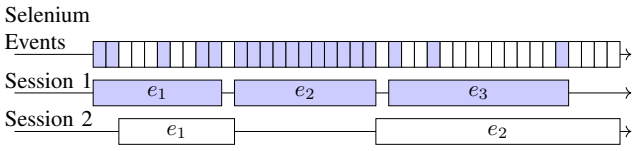

Fig. 2. Execution semantics of low-level and high-level events in sessions: stories specify possible sequences of high-level events that run in parallel sessions (the bigger boxes in the last two timelines). Each high-level event is translated to a sequence of low-level events, and the interleaving is done at the level of low-level events (the first timeline).

to easily capture functional specifications and user stories, and then self-generates business processes and maps, authors, prioritizes executable design documents, and a complete test plan to ensure that the feature, process, or new system is implemented perfectly [17].

At the core of the Provengo tool is Behavioral Programming—a modeling paradigm that enables developers to design complex reactive systems in an incremental and modular manner, where each module is aligned with a single aspect of the system behavior, preferably a requirement [18], [19]. Specifically, the Provengo tool harnesses the power of BPjs [20], a framework for running and analyzing behavioral programs, as its foundational technology. By leveraging BPjs and incorporating test-oriented tools, Provengo empowers testers with a new model-based testing method for creating executable test models using behavioral programming.

One of the key benefits of behavioral programming is that it allows developers to directly align the modules they are creating with the project's requirements [21]. Additionally, the simple execution semantics it provides makes it easy for modelers to run the requirements directly, which can streamline the development process and lead to more accurate and efficient system design.

Provengo employs an event-based programming model where different stories (threads of behavior coded in JavaScript) can request, block, or wait for events. This allows for the precise organization and specification of independent aspects of behavior in separate stories. For example, a story can dictate that an X event (test step) cannot occur before a Y event by waiting for Y while blocking X. Once Y is triggered, the blocking is removed.

Provengo's execution mechanism is depicted in Figure 1. The tool repeatedly runs all stories until they reach a synchronization point that they mark by calling a JavaScript method named 'sync'. At this point, the stories declare the events they have requested, waited for, and blocked as parameters to the 'sync' method. The mechanism then selects an event requested by at least one story and is not blocked by any story, and continues running the stories. This cycle is repeated until there are no more events to trigger. The goal of the model (its semantics) is to define the set of all runs (tests) that are consistent with the requirements.

Example 1. *In testing an application with two buttons, red and green, consider an example where there are two stories. The first story requests to push the green button three times and the second story requests to push the red button ten times. When these stories are run together, the execution progresses by running both stories to their first synchronization point, where one requests to push the red button, and the other requests to push the green button. In this model, there is no blocking, meaning that both events are possible, and one is selected arbitrarily. The process then repeats. The semantics of this system is a set of* 1064 *possible runs. This number can be calculated by* (13 *choose* 3) = 1064*, where* 13 *is the total number of button presses (*3 *green plus* 10 *red) and* 3 *is the number of times the green button is pressed.*

As discussed above, it is rarely the case that all possible combinations of the stories are allowed. The following example demonstrates this observation.

Example 2. *Provengo allows testers to use behavioral programming idioms to add new constraints to their model without modifying the original code. For example, a new rule can be added to prevent the green button from being pressed twice in a row by adding a new story that waits for the red button to be pressed before allowing the green button to be pressed again. This additionally decreases the number of possible test scenarios, in this case, to* (11 *choose* 3) = 165 *tests, as there are* 11 *spaces between the ten presses of the red button to place the* 3 *presses of the green button.*

Provengo allows for specifying tests on various levels of abstraction, including the direct use of Selenium events in stories, for testing web applications However, our examination shows that specifying tests on a higher level of abstraction, using events like "login", "add to cart", or "checkout", is more effective. We found out that, by doing this, it is easier to understand the intent and purpose of the test, and it also makes the tests more resilient to changes in the user interface. This practice is common in other techniques as well, such as in Cucumber [22], where feature files are specified using such abstractions, and the step files define the implementation details of each abstraction. Additionally, using abstraction may lead to better communication and collaboration among team members as the tests are described in a more easily understandable way for non-technical stakeholders. For example, an event like 'login' can be translated into a series of Selenium events like "fill the username in the first input box," "fill a password in the second input box," and "press the Login button." The abstraction allows testers to focus on the functionality being tested rather than the details of how it is being tested, improving maintainability and readability.

The mechanism for integrating high- and low-level events is a process of interleaving, which allows for the coordination of both types of events in a parallel way as illustrated in Figure 2. In this process, the stories specify possible sequences of high-level events that run in parallel sessions. These high-level events are represented as the bigger boxes in the last two timelines. Then, each high-level event is translated into a sequence of low-level events. This translation is necessary because high-level events are more abstract and less specific,

while low-level events are more concrete and contain many details we want to encapsulate. The separation is necessary to make the tests more actionable, more understandable, and more resilient to changes. The interleaving is done at the level of low-level events, as seen in the first timeline. This means that the low-level events from each parallel session are coordinated so that they can run together without conflicts. This process allows for the efficient execution of high-level events and a more detailed view of low-level events.

It is worth noting that sessions in Provengo are not necessarily directly correlated with stories. One story can specify constraints for multiple sessions, such as blocking event X in session one until event Y in session two is triggered. Additionally, multiple stories can relate to the order of events in one session. For example, story one could specify triggering ten X events in a session, while story two specifies triggering three Y events in the same session. This flexibility allows developers to create complex test models by combining and reusing individual components and by specifying multiple constraints across different sessions. These ideas are elaborated and demonstrated in the following sections.

In conclusion, Provengo is a powerful platform that streamlines the software development process by automating requirements gathering and eliminating manual handoffs between the business and development teams. It enables development teams to create process flows and user journeys directly from user stories and functional descriptions, allowing software developers to align to the business understanding fully. Additionally, it automates the functional testing process, generating an exhaustive list of functional tests to ensure comprehensive coverage of the system while improving the performance and quality of the final product.

We only outlined the main features and principles of the Provengo tool to provide a self-contained text. The tool has a lot more depth and capabilities that can be further explored. To gain a complete understanding of the tool and its potential, we recommend readers consult the tool's documentation and gain hands-on experience with it. In the next section, we will provide details about the system that we have tested using the Provengo tool, the tests that we have generated, and the bugs that we have found. This will provide a deeper understanding of how the tool can be used in practice and its benefits.

IV. System Under Test: Magento-2

Magento-2 [23] is an open-source e-commerce platform that enables businesses to set up and manage virtual stores. In 2018, Adobe Inc. took over the platform and has since been responsible for its maintenance. The platform is used by over 100K online stores and holds a 1.72% retail market share worldwide [24]. It includes a central server, database server, backend server, and a web-based interface. The platform provides REST APIs for communication with the server and has two modes of operation: administrator and client.

We next describe the specific part of Magento-2 that we test, namely the purchase process, and explain why it is important to include tests with multiple sessions. The online purchase process typically starts with customers browsing the store website and adding items to their virtual cart by clicking the "Add to cart" button. They can review their cart, make changes, and proceed to checkout, where they provide their shipping information and payment method. The process can vary based on the store and payment methods. Testing concurrent sessions involves simulating multiple users interacting with the system simultaneously, which can improve customer experience by allowing access from multiple devices. However, concurrent sessions make testing difficult due to the possibility of race conditions and conflicts between actions.

V. How is Magento-2 Tested Today? Why Multi-sessions are Not Sufficiently Tested?

Magento-2 is tested using a combination of automated and manual testing methods. The automated testing methods include unit, integration, functional, and performance testing. Load testing and stress testing are typically done using external tools such as Apache JMeter, Gatling, Apache ab, and Siege. Unit testing is done using PHPUnit and JavaScript testing frameworks like Jasmine, while integration testing is done using the Magento-2 Functional Testing Framework (MFTF) [25], which is built on top of the PHPUnit testing framework and Selenium. Performance testing is done using the Magento-2 Performance Toolkit (MPT), which provides tools for generating load on the server and measuring the response time, throughput, and other performance metrics. Acceptance testing is done to ensure that the system meets the business requirements and is ready for release. Manual testing is done to validate the functionality, usability, and performance of the system.

The Magento-2 MFTF testing guidelines encourage testers to use short, granular tests focusing on a single functionality: "Keep your tests short and granular for target testing, easier reviews, and easier merge conflict resolution. It also helps you to identify the cause of test failure [26]." The benefit is that it makes test-suite maintenance easier and helps identify the reason for failed tests. While granular testing has its advantages, it may not always be effective at detecting bugs that surface when the site is utilized in a more comprehensive fashion. To address this, more realistic and comprehensive testing methods may need to be employed, but this can come at the cost of ease of maintenance and debugging.

Here, we explore story weaving as a way to balance focused granularity and comprehensiveness in testing. We demonstrate how developing stories that represent focused aspects of tests and automatically generating comprehensive tests from these stories, allows testers to create tests that are both thorough and manageable while maintaining control over the testing process.

Testing the checkout functionality of Magento-2 using multiple sessions demonstrates well how multi-session, multi-user tests can uncover new bugs. In this scenario, multiple users open separate browser sessions and perform actions such as adding items to their carts, adjusting quantities, and proceeding to checkout. While this simulates real-world scenarios, the current Magento-2 test suite does not include such situations.

This is believed to be due to the recommendation cited earlier that tests should be short and granular, as such tests involve multiple components, multi-session capabilities, and the checkout process. Indeed, without story weaving, it may be challenging to maintain test suites with many possible ways to interleave actions in sessions, as even small changes to the purchase logic would require intricate changes to each of the tests.

Readers may wonder why the random stress tests of Magento-2 did not uncover the bugs we report in this paper. We believe that the reason is that random stress tests lack focus on specific use cases and a clear specification of expected behavior. While they can identify some issues, they may miss relevant ones and fail to provide a comprehensive understanding of the system's behavior. In contrast, story-based testing with Provengo's tool allows testers to focus on relevant issues by creating test scenarios based on realistic use cases and specifying expected behavior. This enables a more effective exploration of the system's behavior and identification of issues that may have been missed by random stress testing.

VI. Our Test Model for Magento-2

This section outlines our approach for specifying test models in the case study. The key finding is that the process of story interweaving enables test development to be done incrementally, yielding quick returns on investment while also allowing for smooth growth. Following an initial step of specifying a set of high-level events to serve as the vocabulary for the tests, test engineers can then write and interweave stories, as we illustrate below.

A. High-Level-Events Specification

The initial step in utilizing Provengo for testing a platform like Magento-2 involves identifying and clearly defining high-level events, which will serve as the foundation for defining the user stories, as demonstrated in this section.

Provengo offers a convenient method, called `defineEvent`, to facilitate the creation of these events, as demonstrated in Listing 1. The code utilizes Selenium commands to interact with the website and execute the tests. The function parameters are the session's class, the high-level event name, and a callback function that represents the sequence of low-level events to be triggered upon the selection of the high-level event. The `defineEvent` function adds the callback function to the session class, giving it the same name as the high-level event name. For example, handling an event called 'Login' within Selenium sessions is achieved by calling `defineEvent(SeleniumSession, 'Login', function(session, data) ...)`. Once the 'Login' event is triggered in a Selenium session, the callback function is executed with the session's object and the event's data as its parameters. The data can be used to provide additional information, for example, the username and password for the login action.

In addition to `Login`, we also defined the `AddToCart` event, which mimics placing store items in the shopping cart, and the `CheckOut` event, which simulates how customers submit their shipping and billing information and confirm their purchase. The latter encompasses tasks such as filling out forms, confirming the purchase, and other activities associated with the checkout process. The complete details of our models can be found in the supplementary material.

Listing 1
A specification of a high-level event called 'Login.'

```
defineEvent(SeleniumSession, 'Login',
    function (session, event) {
  with (session) {
    click('//a[contains(text(),``Sign In``)]')
    write('//input[@id=``email``]', event.username)
    write('//input[@id=``pass``]', event.password)
    click('//button[@id=``send2``]')
  }
})
```

Using these three high-level events and the low-level events that make them up, we could create a test suite for Magento-2 by utilizing story weaving and the Provengo tool. As we will see, this capability helped us uncover new bugs in Magento-2 and simulate real-world scenarios. As we will show, by breaking down the tests into small, focused stories, we can maintain better control over the testing process and make identifying the cause of test failures easier.

B. Story Specification

We started with simple scenarios and increased complexity by adding stories and context. Using test interleaving and automatic story-weaving features, we broke down a standard test script into modular stories and combined them into a powerful test suite. This approach enables efficient and effective testing by focusing on specific system aspects and creating realistic, comprehensive test scenarios. Compared to traditional testing methods, this methodology offers greater testing power and flexibility by allowing testers to move beyond linear scenarios and combine modular stories in a more dynamic way.

We begin with a simple linear purchase story. The story in Listing 2 logs in the user and adds two products to the user's cart (with different options). Next, it removes the second product from the cart and verifies that the first is still in the cart. Finally, it checks out and verifies that the cart includes the expected items after this specific sequence of events.

In many testing frameworks, testers explicitly specify such scenarios. This code demonstrates it using Provengo. However, thinking of all possible test scenarios and specifying them, may be infeasible. To address this, the Provengo tool allows for interleaving tests, as demonstrated in Listing 3. The code defines a list of Magento-2 users and creates a story for each. This story login, adds a product to the cart, and checkouts.

According to the semantics of session interleaving described in Section III, the tool generates all possible orders of events that can occur when multiple users interact with the system simultaneously. Figure 3 shows the state graph generated by Provengo for this model. It only shows the possible interleaving of high-level events with two users. In reality, when the interleaving of low-level events and more users are considered,

Listing 2
A SIMPLE LINEAR PURCHASE STORY.

```
story('My first end-to-end test', function () {
  with (new SeleniumSession().start(URL)) {
    login({username: 'ron@example.com',
      password: 'ron3@example.com'});

    addToCart({ category: 'Women',
      subCategory: 'Tops',
      subSubCategory: 'Jackets',
      product: 'Stellar Solar Jacket',
      options: ['S', 'Blue'], quantity: 3})

    addToCart({ category: 'Men',
      subCategory: 'Tops',
      subSubCategory: 'Jackets',
      product: 'Kenobi ail Jacket',
      options: ['S', 'Blue'],
      quantity: 3})

    removeFromCart({
      product: 'Kenobi Trail Jacket'})

    checkExistenceOfProductInCart({
      product: 'Stellar Solar Jacket' })

    checkOut({
      shippingMethod: 'Fixed',
      verifyItems: ['Stellar Solar Jacket'],
      verifyNot: ['Kenobi Trail Jacket']})
  }
})
```

Listing 3
FIRST EXAMPLE OF SCENARIO INTERLEAVING.

```
const users=[{username: ..., password: ...}, ...]
users.forEach(user =>
 story('Story for ' + user.username, function () {
   with (new SeleniumSession().start(URL)) {
     login(user)
     addToCart({product: 'Stellar Solar Jacket',
        ... })
     checkOut({shippingMethod: 'Fixed', ...})
   }
 }))
```

the graph becomes much larger (too large to fit here). Each path in this graph represents a test that may reveal a bug in the online store being tested. In this simple example, we can run all possible paths automatically to ensure that the system behaves correctly in all the possible user interactions modeled by these paths. In more complicated cases, however, it may not be feasible, and a subset of these behaviors will have to be chosen, as we demonstrated later.

We now demonstrate anti-scenarios and story-weaving constraints. For this, we separate the "Add to Cart" stories from the "Checkout" story, as shown in Listing 4. This simulates a scenario where customers add items to their shopping cart in one session and complete their purchase in another.

An "add to cart" story is created for each session, adding a random product to the cart, and a single "checkout" story is executed in an additional session. All stories use the same user credentials. We permit users to initiate a checkout session while adding items to their cart (we allow them more, as elaborated below). This concurrent session setup creates numerous testing scenarios, as the actions performed during the checkout process can overlap with the actions of adding items to the cart. This offers an opportunity for Provengo's

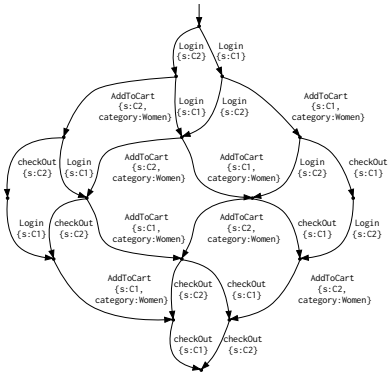

Fig. 3. The state graph of two interleaved stories.

Listing 4
SEPARATING THE SEQUENCES OF EVENTS FOR THE ADDTOCART AND CHECKOUT ACTIONS INTO INDEPENDENT STORIES.

```
const products=[{Product: '...', category: ...}, ...]

for (let i = 0; i < NUM_OF_SESSIONS; i++) {
  story('Add to cart story #' + i, function () {
    with (new SeleniumSession().start(URL)) {
      login({username: ..., password: ...})
      let prod = choose(products)
      addToCart(prod)
    }
  })}

story('Checkout story', function () {
  with (new SeleniumSession().start(URL)) {
    login({username: ..., password: ...})
    checkOut({shippingMethod: 'Fixed'})
  }})
```

testing method to explore a variety of combinations that human testers might overlook.

C. Using Event Blocking to Fix a Modeling Bug

The execution of the tests generated from the above stories resulted in failures. While the checkout button is not-clickable before adding items to the cart, our stories do not enforce it, and such scenarios have been generated. This presents us with an opportunity to demonstrate the use of event blocking. Recall that, in behavioral programming, event blocking limits the interleaving of scenarios by temporarily inhibiting certain events from triggering. Specifically, we added an anti-scenario (Listing 5) specifying that the `CheckOut` event is blocked until the `AddToCart` event is triggered. Traditionally, changing a program or script would require modifying the existing code. However, we could implement this restriction in a new, separate module, that specifies the new requirement using the blocking idioms. This approach allows for incremental changes and a direct correlation between individual requirements and test modules, which is especially useful when developing large test models.

A perceptive reader may have identified a minor flaw in the story shown in Listing 5. It only prevents the checkout process until the start of the add-to-cart operation, which may result in a checkout being initiated before the add-to-cart operation is completed, and the proceed-to-checkout button is still not shown. This can occur in rare cases, but we have resolved it

Listing 5
UTILIZING EVENT BLOCKING TO DEFINE AN ANTI-SCENARIO THAT ADDRESSES A DEFECT IN THE TEST MODEL.

```
story('No checkout before add to cart', function (){
  block(Any('Checkout'), function () {
    waitFor(Any('AddToCart'))
  })
})
```

Listing 6
AN INITIAL POPULATION OF THE CONTEXT DATABASE.

```
ctx.populateContext([
  // Users
  ctx.Entity('ron@example.com', 'User', {...}),
  ...
  // Products
  ctx.Entity('Hero Hoodie', 'Product', {...}),
  ...
])
```

by waiting for an event that indicates the end of the add-to-cart process. Although we have corrected this in our code, we have retained it in the paper's listing for simplicity.

D. Binding Stories to Context to Diversify Tests

This section describes how we broadened and expanded our testing by incorporating context. The context is an extension to BP [21] and, inherently in Provengo, that allows the direct specification of context-dependent requirements. The context of the model is defined as a relational database of context objects called 'Entities'. Effect functions can facilitate automated event-triggered updates to this database. Stories can be bound to a query on the DB, automatically spawning a story copy for each new query response. The linked story copy is automatically removed when the response is no longer valid.

We employed these mechanisms to facilitate the growth of our tests by enabling a dynamic range of products and users with which the stories are associated. The initial population of the context database (Listing 6 populates the database with a list of entities, including users and products). Each user is created with a unique email address as an identifier, a 'User' type, and additional properties (e.g., username, password, and cart and checkout statuses). Each product is created with a unique name, a 'Product' type, and different properties (e.g., category, subcategory, options, etc.). Notably, these sets of users and products are defined at the start of the test. Dynamic changes to the user or product list during the test will cause the related stories to adapt accordingly.

Our experiment's context is updated dynamically as shown in Listing 7. The code demonstrates how updates to the context are made. The `ctx.registerEffect` method attaches a function to an event. When the event is triggered, the function is executed and can update the context accordingly. Here, the effects maintain the checkout process's state and the items in the user's cart. The `checkingOut` flag lets us know if the user is currently engaged in the checkout process, and the list of products in the user's cart will be used to confirm the items that should be displayed on the checkout page (see Listing 9 line 16). The first effect, `CheckOut`, is triggered

Listing 7
THE EFFECT FUNCTIONS THAT DETERMINE HOW THE CONTEXT IS DYNAMICALLY UPDATED WHEN EVENTS ARE TRIGGERED.

```
ctx.registerEffect('CheckOut', function (e) {
  let user = ctx.getEntityById(e.username)
  user.checkingOut = true
})

ctx.registerEffect('EndOfAction', function (e) {
  let user = ctx.getEntityById(e.user.name)
  if (e.eventName == 'AddToCart') {
    if (!user.checkingOut)
        user.cart.push(e.product)
  } else if (e.eventName == 'CheckOut') {
    user.checkingOut = false
    user.cart = []
  }
})
```

Listing 8
QUERIES THAT EXTRACT OBJECTS FROM THE CONTEXT DATABASE

```
ctx.registerQuery('Prod', ent => ent.type=='Product')
ctx.registerQuery('User', ent => ent.type=='User')
```

when a checkout begins, setting the `checkingOut` flag of the user to true. The second effect, `EndOfAction`, is triggered when an event ends. This effect has an if statement that checks the value of `e.eventName` to determine which action has ended. If the event that ended is `AddToCart`, it pushes the `e.product` into the user's cart if the `checkingOut` flag is not set. If the event that ended is `CheckOut`, it retrieves the user entity and sets the `checkingOut` flag to false and empties the user's cart. Note that we only update the cart if the `checkingOut` flag is false, as the requirements dictate that items added to the cart during the checkout process should not be included in the final checkout list.

The code demonstrates the management of dynamic context in Provengo. When testing complex systems, keeping track of objects' states is crucial. In Provengo, tests are defined as sequences of actions rather than reactive systems, so it is not possible to directly query the system being tested for its state. The code shows how Provengo allows maintaining the object's state using the dynamic context.

Provengo also supports the definition of context queries as shown in Listing 8. Two queries are registered—`Prod` and `User`—that return all entities whose `type` is `'Product'` and `'User'` respectively. These queries can filter and retrieve specific sets of data within the application.

The query's effectiveness lies in binding stories to context, i.e., to the queries' answers, as demonstrated in Listing 9. Using just two stories, we detected two bugs in Magento-2. The stories are bound to a query by passing the query name as the second parameter of the `ctx.story` function. This ensures that a separate copy of the story is kept for each answer to the query. In Listing 9, we bind the stories to the 'User' query, meaning they will be executed once for each user.

Incorporating context provides the benefit of keeping a record of objects such as products, users, and others related to the stories. We can associate stories with objects with specific properties, like binding a story to users with non-

Listing 9
TWO STORIES BOUND TO THE 'USER' QUERY.

```
ctx.story('Add to cart', 'User',
    function (user) {
  with (new SeleniumSession().start(URL)) {
    login({user: user})
    let product = getRandomProduct()
    addToCart({product: product, user: user})
  }
})

ctx.story('Checkout', 'User', function (user){
  with (new SeleniumSession().start(URL)) {
    waitFor(Any({eventName: 'AddToCart',
        userName: user.name}))
    login({user: user})
    checkOut({shippingMethod: 'Fixed',
        user: user, verifyItems: user.cart})
  }
})
```

empty carts. This broadens the model's richness, as simple stories can now provide more advanced tests when activated in specific contexts. This makes testing dynamic websites like Magento-2 possible. The model is context-aware, and events can automatically spawn and terminate stories as objects are added or deleted, seamlessly adapting to these changes. Furthermore, recording the system state at any given time presents a significant advantage as it enables the examination of the SUT state test results against the test model, thereby resolving the challenge of creating an oracle [27].

E. Expanding to the Full Magento-2 Store

Up to now, we have incrementally specified a model that we more or less anticipated from the beginning. To test complete systems, one has to be able to dynamically add new areas of behavior that were not planned for. For example, we added scenarios that specify administrator operations on the store and used the tool to examine their interactions with the original model. Specifically, we expanded the model to include two new scenarios: "add new products" and "change price" (see the supplementary material). Generating the state graph for this model using the tool, resulted in a graph that has 5,120 states, compared to 448 in the original model, and 29,952 edges compared to 2,496.

The challenge was to allow existing tests, detailed above, to operate alongside the new activity we defined. While existing approaches would allow independent scenarios to run in parallel to test the system load capacity, our expansions integrated behaviors more deeply. For example, new products were added to the context's list of products at runtime (using effects), automatically allowing users to add them to their cart. We added many tests that combined different requirements and verified that the system worked even in extreme situations that were difficult to formulate through a collection of manual tests.

Our experiment showed that using context to maintain the system state and store information like the site's product list is beneficial. While Provengo's tool allows defining a fixed product list, storing information in context enables adding behaviors that dynamically adapt. This allowed us to add new behaviors without altering the purchase-related behaviors and yet interleave them in exciting ways. Overall, we found that testing systems with complex states, such as a store's item list, shopping cart item list, user list, etc., can significantly benefit from the organized use of context.

VII. TEST SUITES GENERATION AND EXECUTION

This section describes how we ran our models to identify bugs. We recall that a test case is a sequence of low- and high-level events that conform to the specification (i.e., the stories and anti-stories). The Provengo tool allows for running random test cases "on the fly" by using the execution engine to select one requested-and-not-blocked event at a time. Yet, the many possible paths make it impractical to test all of them. To overcome this, Provengo can sample many test paths (without acting on the SUT) and utilize a genetic algorithm (GA) to find test suites that reach high coverage. The selection is based on a ranking function provided by testers, which defines the coverage criterion and assesses the effectiveness of the test suite in detecting bugs in Magento-2. The integration of the GA into the Provengo tool makes the process of implementing the sampling and selection simple and efficient.

We started with the first option of executing random paths using the 'run' command, which randomly selects a path and executes it by acting on the SUT with Selenium. This phase allowed us to develop and debug the test model. However, as expected, we encountered many test failures due to model bugs, leading to a time-consuming process. For instance, we initially forgot to specify that checkout should only be attempted when items are in the shopping cart. Additionally, when testing the single-story model version (Listing 2), we encountered typical Selenium issues, such as incorrect elements' addresses (i.e., xpaths). Despite the time investment, we believe this phase is cost-effective as it helped us clarify what we wanted to test. Including developers and product owners in this phase would be even more beneficial, as it would facilitate communication and help to solidify the testing requirements.

Once we were satisfied with the model's correctness, we used the 'sample' command. This command collects a large number of random paths from the test model that conform to the specified requirements. The gathered tests include Selenium events that were selected during the sampling process; however, the process selects such events without actuating the SUT, resulting in an extremely fast sampling. Once collected, these tests can be executed using the 'run' command. We utilized this feature to gather statistics. Since various factors outside of the test sequences can impact the test results, such as timing and memory availability, some tests may pass on some occasions but fail on others. By repeatedly running the same tests, we identified and resolved these issues, for example, by waiting for the end of add-to-cart events instead of just waiting for their beginning. This helped us improve the reliability and accuracy of our testing results.

Next, we used Provengo's optimization engine to select the most relevant tests based on a specific criterion. We used it to choose a small group of tests encompassing a wide range of behaviors, which will be shown later in the paper.

TABLE I
NUMBER OF FAILURES PER NUM_OF_SESSION (MARKED BY #).

#	Add-to-cart sessions	Failures
1	One parallel session	100%
2	Two parallel sessions	98%
3	Three parallel sessions	97%
4	Six parallel sessions	96%

TABLE II
PERCENT OF FAILED SCENARIOS FOR EACH SCENARIO TYPE, VALIDATING OUR HYPOTHESIS REGARDING THE SEQUENCE THAT TRIGGERS THE BUG.

#	Ordering	Failure
1	`Login` before `AddToCart`	100%
2	`Login` after `AddToCart`	0%

The command for this analysis is 'ensemble,' requiring two inputs: the desired ensemble size and a ranking function for evaluating ensemble quality, that assigns a score to each potential ensemble. The GA, integrated within the tool, uses the scores to choose the optimal ensemble of the specified size with the highest ranking score. This guarantees that the selected ensemble of tests offers the best-found coverage of the intended scenario.

VIII. TWO BUGS FOUND IN THE MAGENTO-2

The section describes how we analyzed the test results and gives a concrete example of two bugs that were found and reported to the development team of Magento-2. We sorted the errors detected during runs. At first, most of the errors were due to mistakes in the model. After fixing the model, the real bugs are sorted according to the phenomena they raise. For each type of error, we characterized the buggy executions for the development team to quickly reproduce the problem and plan how to fix them (e.g., which teams should be sent the problem? What is its level of importance? etc.)

A. The 'Empty Cart' Bug - Issue #37465

We discovered[2] a glitch in the shopping cart while testing. It would sometimes appear empty at checkout, even after adding products. Our initial effort in comprehending the underlying cause of the issue was to accumulate sufficient tests, both with and without the bug, to facilitate the examination of the distinct attributes of the test sequences that triggered it. One of our experiments, reported in Table I, was to alter the number of parallel add-to-cart sessions (see Listing 4) to create a simulation of adding items to the cart from many concurrent sessions in parallel and a single session from which a user logs in and checks out the items. As the table reveals, almost all tests failed when we ran scenarios with one parallel session. This challenged us as it prohibited us from evaluating the differences between faulty and non-faulty tests. Fortunately, after testing several parameters, we discovered we could attain the desired diversity by increasing the number of parallel add-to-cart sessions, as illustrated in the table.

To classify and determine the source of the inconsistent behavior, we examined scenarios with and without the error. We discovered that in all failed scenarios, the `Login` of the `CheckOut` story was selected before the first `AddToCart` event. In other words, the issue we revealed is that when the login for the checkout session is completed, and the cart is empty, the system does not refresh the cart in this session and does not show the 'proceed-to-checkout' button even when items are added to the cart. As a result, our test fails when attempting to press this button, which was expected to be enabled due to the items already added to the cart. Some readers may rightfully say that this is not a bug and that we should have refreshed the page before pressing the button. Nevertheless, we argue that in the case of the Magento-2, this is a bug since the cart status (e.g., number of cart items) is synchronized among the different sessions, and only the 'proceed-to-checkout' button is not. The results of the tests we conducted to validate our conclusion are displayed Table II.

Although successful, our experience with this bug also highlights some limitations of the Provengo tool. Specifically, when we encountered a very low probability of missing the bug, we had to consider adjusting parameters to change the outcome. It would be beneficial if the tool could support this approach, e.g., by changing the event-selection distribution. Additionally, when we had a collection of passed and failed tests, we manually analyzed them to determine the source of the bug. This process can be automated using process mining algorithms, such as alpha-miner [28], or automata-learning algorithms, such as L^* [29] or RPNI [30]. These algorithms can effectively analyze the set of tests and identify patterns, relationships, and anomalies that could indicate the root cause of the bug. Automating this process reduces the time and effort required to identify bugs, making the bug-debugging process more efficient and effective.

B. The 'Shipping Fee' Bug - Issue #37466

In our testing, we identified[3] a rare occurrence where the shipping fee calculation was incorrect. More specifically, a calculation error caused the system to become inoperable when the customer attempted to place an order. As a result, the confirmation screen, which indicates the successful completion of the transaction, was not displayed.

Encouraged by the "empty cart" bug, we wanted to increase the frequency of occurrence by adjusting the testing parameters. However, our efforts were unsuccessful when we tried to apply the same methodology to the second bug. As a result, we expanded our methodology and tools and developed a new approach called "bug amplification," which proved effective in resolving the issue.

C. The Bug Amplification Method

The amplification method aims to enhance the likelihood of replicating a bug. Initially, the bug was only triggered

[2]https://github.com/magento/magento2/issues/37465

[3]https://github.com/magento/magento2/issues/37466

TABLE III
THE PARAMETERS USED IN THE BUG-AMPLIFICATION PROCESS.

Parameter	Value
Sample size	10,000
Ensemble size	15
Number of ensembles used	10
Number of cases found	14
Runs of the 14 cases	64
Total cases failed	369
Percentage of total cases failed	41%
Control group size	1,000
Percentage of failed in control group	3.4%

with a low probability by simulating two users simultaneously adding three items to their shopping cart and completing the checkout process. The difficulty was to increase the probability of encountering the issue. We utilized the ranking function to discover more bug occurrences, as outlined in [31] and discussed in Section VII. We created a sample of 10K test scenarios. We created a small subset, known as the ensemble, that satisfies a criterion we correctly assumed related to the probability of detecting the bug. We executed the ensemble against Magento-2 and managed to derive a small set of tests with a high likelihood of failure. The ranking function that we used was: "Count the number of tests in which: (1) the event `AddToCartStory` for the same user appears in the time frame between `Checkout` and `WaitForVisability` of the message 'Your order number is:'; and (2) the temporal distance between these two events is smaller than a specified constant."

The inspiration for utilizing this type of ranking function originates from [31], which introduced the notion of test coverage, considering that overlapping concurrent operations may be potential sources of bugs. In our situation, we hypothesized that the overlap between `AddToCart` and `Checkout` was responsible for the bug. Therefore, we prioritized the addition of tests with such overlaps, awarding ensembles that include them with higher ranks. We implemented this function and utilized the `ensemble` command to generate a highly-ranked test suite. We then executed the generated ensemble and monitored the system to identify the manifestation of the bug. We repeated this procedure until we found 14 test cases where the bug emerged with a reasonably high chance.

The specific details of the amplification process are provided in Table III. Our initial sample contained 10K cases, from which we generated ten ensembles that identified 14 failures, each comprising 15 test cases. We ran each failed test 64 times and obtained 369 failures, accounting for 41% of the total runs. Figure 4 illustrates the number of failures recorded for each of the 14 tests we collected. The graph indicates that most of them produced failures with a sufficiently high probability to warrant inclusion in a report to the development team.

To verify our findings and bug-amplification method, we executed 1,000 random tests to confirm they were not due to chance. The outcomes revealed that the issue arose randomly in only 3.4% of the cases, compared to over 41% for the tests we were able to generate.

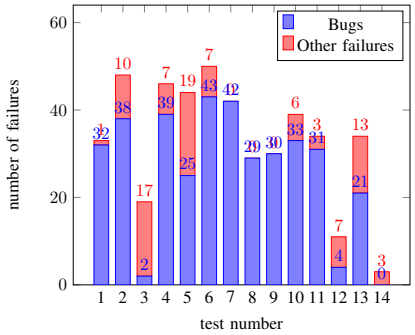

Fig. 4. The number of failures for each test out of 64 runs. The blue bars indicate how many times each test triggered the "shipping fee" bug, while the red bars indicate the number of other bugs that have not yet been studied.

IX. SUMMARY OF LESSONS LEARNED

We demonstrated how Provengo's scenario-based testing advances model-based testing in small steps. By using the idioms of blocking and requesting in scenario-based programming, system stories can be broken down into components that can be assembled to automatically generate tests. This demonstration provides the following answers to our research questions, listed in the introduction:

RQ1: Is it possible to quickly initiate an effective testing process using the scenario-based modeling approach supported by the tool?

A: Yes, we initiated a simple model that we later enhanced to richer models while continuously generating meaningful tests.

RQ2: Can modeling be done without requiring knowledge of formal methods?

A: Yes, we only used simple models that resemble standard scripts used in test automation.

RQ3: Is it feasible to apply model-based testing on an existing system without an explicit requirement document?

A: Yes, used no requirement document, just common sense.

RQ4: Can the model be used to generate test suites that cover specific areas of interest?

A: Yes, we demonstrated how ranking functions target the test suites to specific types of tests.

RQ5: Is it possible to discover new bugs in a real system using this approach?

A: Yes, we found and reported bugs in a well-tested system with many users and developers.

In our opinion, the most beneficial direction for tool development is better support for context management. Specifically, we learned that expanding a test model to multiple areas requires efficient use of dynamic context changes. We suggest expanding tools that make it easier for testers to define such models. We also propose to strengthen bug detection and amplification tools that proved helpful in our examination.

References

[1] H. Gurbuz and B. Tekinerdogan, "Model-based testing for software safety: a systematic mapping study. softw. qual. j. 26 (4), 1327–1372 (2017)."

[2] L. Apfelbaum and J. Doyle, "Model based testing," in *Software quality week conference*. Citeseer, 1997, pp. 296–300.

[3] B. Elodie, A. Fabrice, L. Bruno, and B. Arnaud, "Lightweight model-based testing for enterprise it," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 224–230.

[4] V. Garousi, A. B. Keleş, Y. Balaman, Z. Özdemir Güler, and A. Arcuri, "Model-based testing in practice: An experience report from the web applications domain," *Journal of Systems and Software*, vol. 180, p. 111032, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001291

[5] "How to improve your workflow using model-based testing - freecodecamp," https://www.freecodecamp.org/news/improve-your-workflow-using-model-based-testing/.

[6] M. Taromirad and R. Ramsin, "Mbt in agile/lightweight processes: a process-centred review," *IET Software*, vol. 13, no. 5, pp. 327–337, 2019.

[7] J. Boberg, "Early fault detection with model-based testing," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, 2008, pp. 9–20.

[8] L. Villalobos-Arias, C. Quesada-López, A. Martinez, and M. Jenkins, "Model-based testing areas, tools and challenges: A tertiary study," *CLEI Electronic Journal*, vol. 22, no. 1, pp. 3–1, 2019.

[9] B. Robbins, "Characterizing software test case behavior with regression models," in *Advances in Computers*. Elsevier, 2017, vol. 105, pp. 115–176.

[10] M. Bernardino, E. M. Rodrigues, A. F. Zorzo, and L. Marchezan, "Systematic mapping study on mbt: tools and models," *IET Software*, vol. 11, no. 4, pp. 141–155, 2017.

[11] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[12] M. Masud, M. Iqbal, M. Khan, and F. Azam, "Automated user story driven approach for web-based functional testing," *International Journal of Computer and Information Engineering*, vol. 11, no. 1, pp. 91–98, 2017.

[13] M. Bar-Sinai, A. Elyasaf, A. Sadon, and G. Weiss, "A Scenario Based On-Board Software and Testing Environment for Satellites," in *59th Israel Annual Conference on Aerospace Sciences, IACAS 2019*, vol. 2, 2019, pp. 1407–1419.

[14] ThoughtWorks. (2023) Selenium. [Online]. Available: https://www.selenium.dev/

[15] G. Candea and P. Godefroid, "Automated software test generation: some challenges, solutions, and recent advances," *Computing and Software Science: State of the Art and Perspectives*, pp. 505–531, 2019.

[16] "Front page - provengo technologies," https://provengo.tech/.

[17] "Provengo linkedin," https://www.linkedin.com/company/provengotechnologies.

[18] D. Harel, A. Marron, and G. Weiss, "Programming coordinated behavior in java," in *ECOOP 2010 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2010, pp. 250–274. [Online]. Available: https://doi.org/10.1007/978-3-642-14107-2_12

[19] ——, "Behavioral programming," *Commun. ACM*, vol. 55, no. 7, pp. 90–100, 2012. [Online]. Available: https://doi.org/10.1145/2209249.2209270

[20] M. Bar-Sinai, G. Weiss, and R. Shmuel, "BPjs," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3270112.3270126

[21] A. Elyasaf, "Context-oriented behavioral programming," *Inform. Software Tech.*, vol. 133, p. 106504, May 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106504

[22] M. Wynne, A. Hellesoy, and S. Tooke, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[23] "Adobe commerce: Powered by magento-2," https://magento.com, 2023.

[24] I. Al Jassasi and Y. Baghdadi, "Categorization of the requirements of social commerce platforms," in *2022 International Conference on Information Technologies (InfoTech)*. IEEE, Sep. 2022. [Online]. Available: https://doi.org/10.1109/infotech55606.2022.9897113

[25] A. Inc. (2022) Introduction to the functional testing framework. [Online]. Available: https://developer.adobe.com/commerce/testing/functional-testing-framework/

[26] A. inc. (2020) Best practices. [Online]. Available: https://devdocs.magento.com/mftf/v2/docs/best-practices.html

[27] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[28] W. Van Der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011, vol. 2.

[29] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540187900526

[30] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," in *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific, 1992, pp. 49–61.

[31] A. Elyasaf, E. Farchi, O. Margalit, G. Weiss, and Y. Weiss, "Generalized coverage criteria for combinatorial sequence testing," *arXiv preprint arXiv:2201.00522*, 2023.