Mutating Test Scenarios

Explore the unknown by disregarding order and welcoming events

Baubak Gandomi

Baubak@Gandomi.com

“It bothered him that the dog at three fourteen (seen from the side) should have the same name as the dog at three fifteen (seen from the front)”

JL Borges (Funes the Memorious)

Abstract

Most testing, whether automatic or manual, strives to raise predictability. This is usually achieved by:

- Following a predefined set of steps
- Removing ambiguity in the findings by eliminating static

In this paper we make the point that these assumptions originate from manual testing and that blindly following such assumptions in automated testing will not provide any additional advantages apart from increasing the processing capacity.

Scenarios that follow a predefined set of steps are too simplistic, and they should be leveraged to go beyond their original intent. In other words, the purpose of test automation should be to perform tasks that cannot be done by manual testing.

We therefore propose a new testing method, which we define as “mutational” testing. In this method, the given scenario can, when needed, adapt, change its structure and order, i.e. “mutate”, to address the challenges that are imposed on it. We also show that by injecting selected “noise” in the execution of a scenario, we open new opportunities in test domains.

We will focus here on two major domains, each an aspect of mutational testing:

- Events: Events taking place during the execution of tests
- Permutations: The user may take a different path than originally intended

The mutational test methods help solve problems such as:

- Software Migration testing
- Software Upgrade testing
- Chaos testing
- End-User testing

Biography

The author, Baubak Gandomi, studied at the Mid-Sweden University, Universität Leipzig (Germany) and finally graduated from the university of Stockholm, Sweden with a degree in Computer Sciences.

He is currently a test automation architect at Adobe, a position he has held for the last 10 years. Apart from upgrade tests he is very interested in cross-product/teams testing and finding measurement methods for assessing functional and transactional coverage.

Baubak is currently residing in France with his family. In his free time, he practices and teaches the martial art of Ninjutsu.

1 Introduction

A lot of this work originates from addressing the complexity of testing systems and product upgrades. After each major upgrade, testers are required to check that certain workflows are still functioning correctly.

Product upgrades can be in many shapes and forms. At Adobe Campaign the notion of “upgrades” involves testing the upgrade of our product from one version to a newer one. Due to the complexity of our product, it could involve API, database and simple functional changes. Our goal was to ensure that processes and workflows starting before the upgrade could be successfully fulfilled after the upgrade.

Like many good ideas, its implementation, and subsequent presentations led to additional questions and problems.

This paper presents our attempts at creating a problem statement that properly represents these problems. It also shows how we have tried to leverage the concepts introduced by the Phased Tests for addressing these problems.

1.1 The Original Problem Statement

When dealing with our original problem, testing upgrades, we usually had two major problems:

- How does one make sure that the problem is related to the change, and is not inherent to the product?
- How can a problem be clearly identified and reproduced?

This led us to identify two underlying issues with the testing we were doing:

1. The Data problem
2. The Time problem

1.1.1 The Data Problem

Traditional testing is insufficient because it often does not consider the data prior to the system change. Systems performing such tests usually manage a database that is regularly updated. Managing such databases is quite complex and is hard to scale. We identify the following problems:

- Mapping data to a specific test scenario is complex
- Mapping data to a specific product/application version is complex
- Updating the data is cumbersome

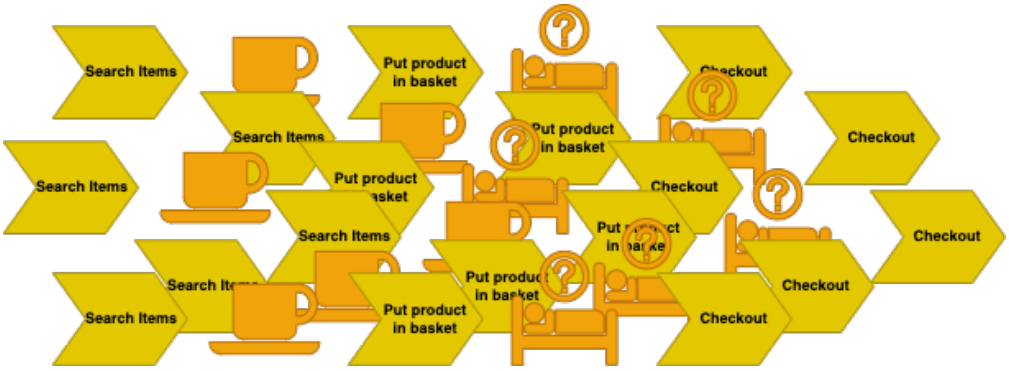
1.1.2 The Time Problem

Most test scenarios imagine a static system. They will perform a set of actions and verify that the end result is what was expected. There are however, two problems with the traditional approach:

Scenarios are not atomic in real life: User Transactions are not usually atomic. The way we use services is rarely a sequence of events with a predefined time interval between each action.

Users rarely perform a series of actions in one go, and they interrupt the process for various reasons

Other scenarios are usually being executed in parallel: You are not the only person executing a scenario / transaction. It is quite likely that others will be performing the same workflow, and may be at different stages.



1.2 Phased Testing: A Solution to the Upgrade Problem

We developed a notion called Phased Testing to address these problems. In that solution, we first put forward a notion of a scenario as a template, which is central to this paper.

Phased Testing was based on two major concepts:

- A test scenario should be interruptible at any point
- A test scenario is declared only once, but we should be able to execute it with all the possible interruptions it may be subject to

When executed in the case of an upgrade test, the system will detect all the possible interruptions a scenario can have. It will then re-execute the scenario as many times as there are possible interruptions, stopping the scenario at the expected interruption point. It will then await the system change, only to carry on the scenarios where they left off.

This mechanic is called “Shuffling”.

2 Lessons Learned from Phased Testing – Enter Mutations

During the various presentations on the subject of Phased Testing and in the course of various development initiatives, we came to certain conclusions.

2.1 An Upgrade is just Another Event

Not all upgrades are so invasive as to interrupt the work on a product. In many cases this can be done with no downtime. In addition, if we want to look at other use cases, we need to look beyond upgrades as a phenomenon. Therefore, we came to the following conclusions.

- An Upgrade can be categorized as a System Change, which can itself be classified as an Event which causes an interruption
- Not all events are interruptive, nor are all they only “upgrades”. We should be able to measure their impact as well

2.2 Scenario Orders are Illusory

We always expect a scenario to follow a certain order of steps. In reality, there is no rule for a user or process to follow a specific order of steps. We tend to write scenarios to reproduce a sequence of events. However, a lot of bugs come from users taking unexpected paths.

2.3 Erasing Walls of Time Dimensions in Test Scenarios

We need to acknowledge that executing a scenario is just a snapshot of what one user will do at a given time to achieve a goal (aka use a functionality). However, we also need to acknowledge that there will be other users achieving the same goal, and who are simultaneously at a different step along a scenario. When an event occurs, the users may be affected differently by the change.

With “The Time Problem”, we acknowledge that an event can occur at any time along the scenario process, and that we will trace these paths so that we can correctly predict the effects of the events on the users.

2.4 Mutations as a Generalized Set of Conclusions

These ideas finally led to a new, generalized set of deductions:

- A scenario is a template that can, and should, be executed with different characteristics
- A scenario can be subject to different Events along its execution. Some are interruptive, some are not
- Scenario steps do not need to follow a predefined order

A scenario needs to adapt, change structure and order to correctly cater to requirements. In other words, we say a Test Scenario can “Mutate”. Even though a scenario can mutate, it still remains a set of steps achieving a goal. In the next chapter we will look more closely at what these mutations can be.

3 Approach

This paper will explore different types of Mutations, and how they can be implemented and used. As mentioned in the introduction, one of the goals of this paper is to leverage the concepts introduced by Phased Tests to enable Scenario Mutation. In this chapter, we will describe the mechanisms that permit us to implement the mutations to which a scenario can be subjected.

3.1 A Scenario and a Set of Steps

In order to allow tests to mutate in the way we want, our scenarios need to have steps that are clearly separated and have clear boundaries.

In many testing tools, such as codeless systems and Cucumber, there is a clear identification of steps, whereas in frameworks originating from unit testing, the notion of a defined step is not naturally implemented.

In the case of Phased Tests we opted for the Scenario as a Class and the steps were represented as Class Methods. Hereafter we will continue following that model.

3.2 Defining a Test Execution Mode

By default, a Test Scenario will follow the order of steps as defined by the developer.

We will provide the possibility for the user to change this behavior at execution time, so that a Scenario will mutate and diverge from the original scenario.

3.3 Test Shuffling and Shuffle-Groups

The concept of “shuffling” involves the multiple re-executions of a scenario, based on a stimulus or a requirement. In the case of Upgrades, the shuffling is based on the possible interruptions a scenario can be subject to whenever an upgrade happens.

Each re-execution or iteration is identified by what we call a Shuffle Group. The Shuffle Group also acts as a context in which the steps have a relationship and share context variables.

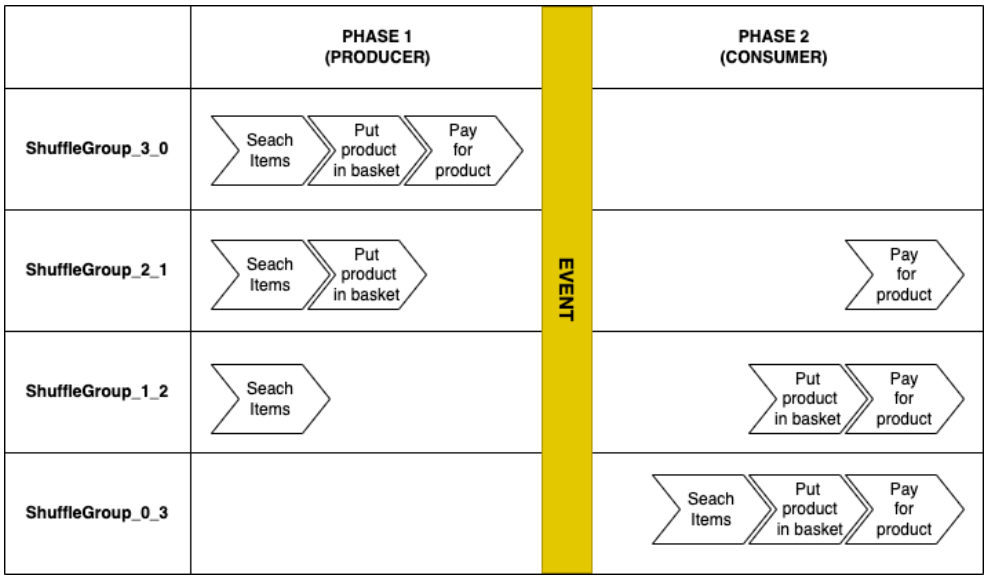


Figure 1 - Shuffle Groups and Phases

At each shuffle phase group, we stop the scenario at a different step. After the “Event”, for each Phase Group the scenario carries on where it left off. Figure 1 shows the shuffle groups in the case of an interruptive event. We divide the execution into two parts called phases, based on their execution relative to the event.

3.4 Producing Data and Consuming Data in Steps

In the context of Phased Tests, the scenario can be interrupted after a given step. This means that the test program is actually interrupted, and as a result the variables will no longer be in the memory. We need to store the values of the variables in long-term storage so that the scenario can carry on after it restarts.

When we “produce” data, it is stored in the context of its phase group. This means that any following step can access this data. Even if we have an interruption, the data is stored in a cache for later use.

We therefore introduced the notions of “produce” and “consume”. A step will produce data that is consumed by later steps.

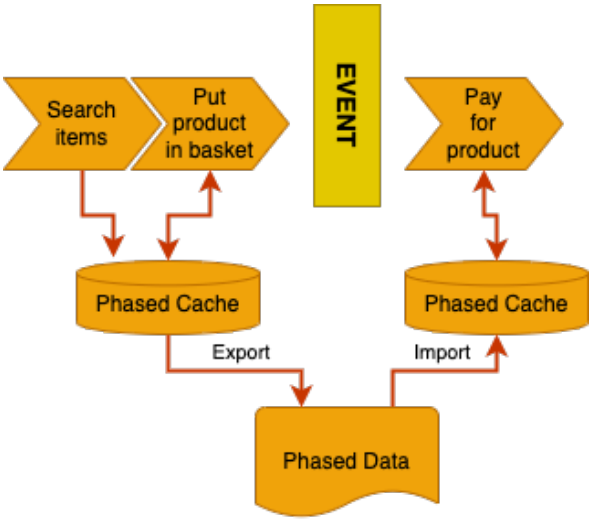


Figure 2 - Storing and maintaining the context between steps and interruptions.

In Figure 2 above, we have represented how produced and consumed data is managed, even after we have interrupted the system to perform our change.

3.5 Test-Driven Mutations

When choosing to mutate, the scenario adapts to the mutation it is exploring. We call this a test-driven mutation as we manipulate the test scenario rather than the event or data. The events for example are simply triggered according to the test scenario steps.

4 Scenarios that Mutate

The notion of mutational testing is that the scenario is a simple executable scenario, which, when needed, can mutate and change its behavior.

As mentioned in the introduction, the mutational scenario can be used as a template to explore new aspects of your product. In this paper, we identify the following mutations:

- Data
- Events
- Permutations

4.1 Data-Driven Mutations

The most simple and common mutation is what is known as “data-driven” testing. In this mode the tests and scenarios are iterated based on a dataset. Typical examples of data mutations are:

- Data Providers
- Language and Encodings

Data-Driven Mutations will be covered summarily as it is a common practice in testing, and there is considerable documentation on this topic.

4.1.1 Changing Test Data

This is not too uncommon as we sometimes need to execute the same scenario with different test data and inputs. This is either done by duplication of the scenario, or by using tools provided by the test framework.

Most test frameworks provide such capabilities, where a scenario will be re-executed for a given set of test data, iterating over each entry. Examples include:

- Parameterized tests in Junit
- Data providers in TestNG
- Data Tables and Scenario outlines in Cucumber

4.1.2 Languages and Encodings

It is often a requirement for many software functionalities to work in multiple languages, and to cater to different character encodings. This means that a test needs to be able to be executed multiple times based on the languages your functionality will be available in.

When you use random data for your tests, it is quite simple to change the encodings or the language of the test data. There are libraries such as Faker where test data can easily be switched between languages and encodings.

4.2 Event-Driven Mutations

A consequence of our previous work on Phased Tests was that we generalized the approach by recognizing that what drives the test is an event.

Events can happen at any time along a scenario. This makes the reproduction and the full impact assessment of an event tricky. In order to create a reproducible and verifiable approach we can state that an event can happen Before, During, and After a scenario step.

With events we can identify two categories:

1. Interruptive Events
2. Non-Interruptive Events

4.2.1 Interruptive Events

Scenarios are rarely an atomic set of steps. Instead, people will perform actions according to whim and will abruptly interrupt the process. During the interruption, many events may happen; the system may be shut down or upgraded. We need to make sure that such major interruptions do not interfere with processes that have not yet come to an end.

Due to the nature of interruptive events, there is a N to 1 relationship between the scenarios and the event. We identify the following use cases when it comes to interruptive events:

- Database Migrations and Changes
- Cloud Migrations
- Major Software Upgrades or Migrations

The Phased Testing framework was originally devised for Interruptive Events, i.e. you need to stop a system so that you can perform some system change such as an upgrade. Once the upgrade is done, we expect that the users can carry on with what they were doing.

The execution of steps in interruptive events is divided into two phases depending on their execution relative to the interruptive event. The phase before the event is called “producer”, because the steps executed before the event produce data used after the event has taken place. Similarly, the phase after the event is called “consumer” because the steps rely on data created in the phase before the execution of the event.

4.2.2 Non-Interruptive Events

In many modern systems, upgrades and system changes are performed with no downtime. This means that events will happen at any given time.

Another major domain that is covered by non-interruptive events testing is testing “resilience”. This means that they need to withstand events. A good example of this is a driverless car challenging a

intersection. The goal of the scenario is for the car to successfully pass the crossroad. The events could be anything from a reckless pedestrian or car. We will have to adapt to the event and then finish the task of driving through the intersection.

As in the case of interruptive events, we never know when the event can happen, so we need to be able to simulate the event happening at all stages of a scenario.

The challenge here is reproducing a detected problem, and to identify the combination that causes a problem.

One may correctly suggest that we can simply add an event within a scenario. The problem with this assumption is completeness, as this will work for one specific step, it will not be efficient if we want to test the effects of an event along the whole process of the scenario. As in the case of interruptive tests, “shuffling” solves this problem. Shuffling will allow us to see the effects of an event at every step of a scenario.

The notion of non-interruptive events is quite close to chaos testing. While most Chaos Testing systems control the triggering of the events, we have opted to have a tight coupling between an event and the scenario. We will not examine if one solution is better than the other, as our focus is on seeing how far our approaches, as described in Chapter 3, can take us.

An event, as we have modeled it, will happen:

- Before a step starts
- During the execution of a step
- After the execution of a step

The challenge here is to be able to avoid slippage, or ambiguity between a step and the event. An event should not “slip” into a step it is not supposed to affect.

An event happening during a step needs to start after the step starts and has to be finished before the next step starts. An event happening between steps needs to finish before the following step starts.

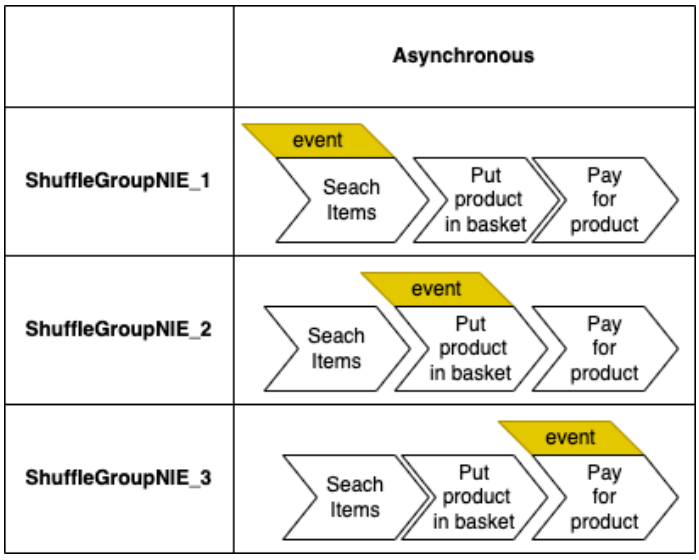


Figure 3 - Shuffling Tests during Non-Interruptive Events

This can be seen in Figure 3 we shuffle the scenario based on the step where we plan to inject the event. Each of these re-executions is identified by its “shuffle group”.

4.2.2.1 Designing and Executing an Event

In our design it is the role of the framework to execute the event. This allows us to have better control of the orchestration and timing of the event in relation to the steps.

This requires events to follow a set of characteristics. We should have the following capabilities:

- Start the event
- See when an event has finished
- Access state information on the event

4.3 PerMutations – Changing the Order

If we take the metaphor of a journey, there are many ways of looking at the process. The most obvious one is to reach a destination by the shortest path. Another way would be to take the more scenic path to get there. Both lead to the same destination.

As in the journey, we usually expect a series of steps leading to a destination. However, this is rarely predictable, and users typically take different paths in order to reach a specific goal.

If we consider that the purpose of the scenario is to reach an end-goal, the steps leading to the end-goal may assume a different meaning. It could be that the order of steps and the paths may become the test itself.

We believe that for a given scenario, we can identify a series of permutations, where some of the steps can be switched. In other words, the steps in a scenario need not be executed in the original order.

4.3.1 Permutating a Scenario

The way we calculate permutations is based on two ideas: the definition order, and the dependencies between steps.

In Phased Tests, steps can communicate variables by storing them in their context. To store a variable, we use a method called “produce”, and the following steps can fetch the variable by performing “consume”. This creates an interdependency between steps, where steps that consume a scenario need to come after one that produces the variable.

Let us take an example as seen in Figure 4 below. We now add a new step to our web shop example: logging in.

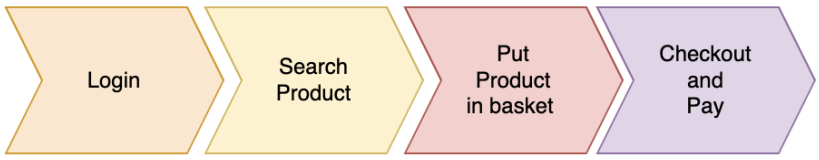


Figure 4 - A revised used case with authentication

If we look at the produce and consume scheme, it will yield a process of the type illustrated in the figure below.

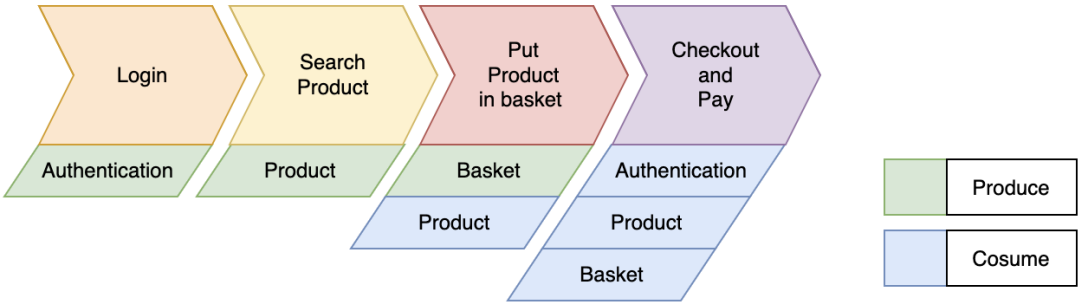


Figure 5 - The produce and consume schema for the web-shop scenario

In this model we make the following assumptions:

- We can search for a product on a web shop and add it to our basket at any time
- A payment requires a user to be authenticated
- Before paying, we make sure that we have selected the correct product

If we look at all the permutations, we can see that the steps “Login” and “Search Product” can be switched. However, Search Product needs to be before “Put Product in Basket”. We also see that “Login” can also happen at any step before the last one since performing the actual transaction requires authentication. If we permutate this scenario we obtain the following cases as seen in Figure 6 below.

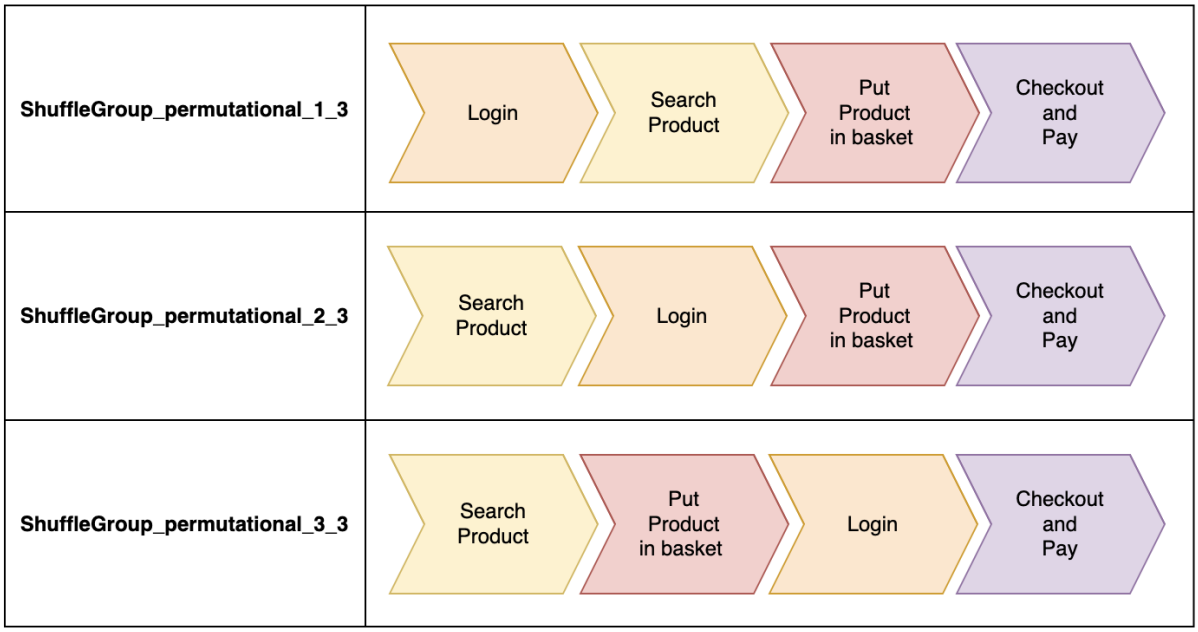


Figure 6 - The permutations of our scenario

4.3.1.1 Limitations and Workarounds

A big problem with this approach is that the notion of “produce” and “consume” is not enough to give a clear view of the order of execution of the steps. Some methods will make changes to an object or even to the system, causing a state change which is a prerequisite in following steps. These changes may be required for the execution of next steps.

To resolve these kinds of issues, one must explicitly produce and consume these resources in the steps, so that the framework can correctly create the permutations. This can be done by introducing these state requirement checks in your steps.

4.3.2 Calculating the Permutations – A Deep Dive

As mentioned earlier, we look at the internal dependencies between the steps in order to calculate the possible permutations of that scenario. This is made possible because of the cross-step communication mechanism we described in Chapter 3.4.

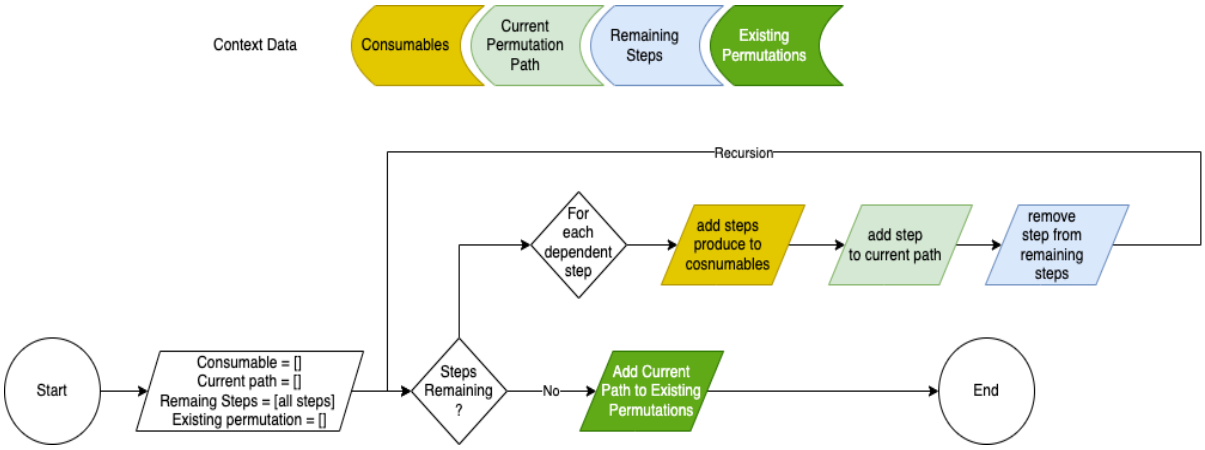


Figure 7 A flow chart describing the permutation calculations.

The algorithm performs the following steps

- Find the steps that can be executed with the given set of produced data.
- For each step start a recursion:
    - Add the data it produces to the set of consumables
    - Add the step to the current permutation path
    - Remove the step from the non-treated steps
- If there are no steps to treat, add the current permutation path to the stored permutations list

5 Conclusion

This work is the culmination of a simple project where we simply wanted to make sure we covered some use cases in upgrades. We believe that with mutational tests, automation tests will step beyond simple non-regression testing and will allow the finding of new paths for testing software. For example, with how events are addressed we provide users the ability to simply test hypothesis regarding the products stability and reactions to external stimuli. We hope this allows us to be more playful with the products we test, and to explore new ways to break the system we test.

The notion of Permutations is something we have been very interested in and have taken time to implement. In hindsight, we have also realized that the notion of permutations puts a greater responsibility on the test steps, as they need to function as independent and self-aware units. This means that they need to have enough information to let us know if they can be executed in a given context.

This problem is not something we usually deal with since scenarios have traditionally been atomic entities, whose steps were not originally intended to be separated. Despite this, we think that in the future such approaches will be more common. Many BDD frameworks are similar in their structure, and the step of writing a self-aware step is not too far.

As a follow-up we would like to see or implement these initiatives in BDD and Low-Code systems, where steps are naturally implemented.

Another potential for the notion of permutations and self-aware steps is if we break the boundaries of the scenario as a unit, and instead create new scenarios from the steps of different scenarios. This enables the discovery of new test paths that, as testers, we had not thought of, and which the users are quite likely to use. Permutations beyond the steps of one scenario are technically possible and would make for a very interesting case study.

Acknowledgements

The author of this paper would like to express gratitude to the following people (in alphabetical order):

- Charles McBride, for his kind and helpful remarks.
- Gary White, who yet again helped me with proof-reading and provided great insights.
- Nemanja Todorovic, for his great suggestions.

References

Gandomi. 2022. “Phased Testing: Testing Systems Undergoing Change” PNSQC 2022 Proceedings: The Evolution of Quality: 40-56.

Cédric Beust, Hani Suleiman. 2007. “*Next Generation Java Testing: TestNG and Advanced Concepts.” Addison-Wesley Professional*

Marty Lewinter, Jeanine Meyer. 2015. *Elementary Number Theory with Programming. Wiley*

Baubak Gandomi. 2021. ”Automated Upgrade Testing: A Process to Test and Validate Software Upgrades”. Adobe Tech Blog, entry posted March 2021. https://medium.com/adobetech/automated-upgrade-testing-a-process-to-test-and-validate-software-upgrades-349e647f34f4 (accessed March 11 2021).

The Phased Testing TestNG Implementation. https://github.com/adobe/phased-testing.

Netflix Technology Blog. 2011. “The Netflix Simian Army”. Netflix Technology Blog, entry posted July 2011. https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (Accessed May 2023)

Paul Marshall. 2022. “Chaos Testing Stardog Cluster for Fun and Profit”, StarDog Labs Blog, entry posted July 13th 2022. https://www.stardog.com/labs/blog/chaos-testing-stardog-cluster-for-fun-and-profit/