Introduction

Many organizations have automated test execution, a critical aspect of test automation. But until they can automate the whole process, it's difficult to keep up with the frequent release cycles associated with DevOps, continuous delivery, or agile. We designed our Digital Automation Intelligence Suite to automate the entire testing pipeline and deliver business-level results.

Eggplant AI is the brain behind the Digital Automation Intelligence Suite. While executing prewritten test cases is commonplace, Eggplant AI focuses on the creation of the test cases themselves — so teams don't have to manually plan and laboriously build a set of static test cases.
Continually running the same set of regression tests only covers a small part of the possible set of tests that could be run. It's also not the best use of resources — a QA developer who's only testing paths that he or she considers won't be able to discover new bugs or problems.

Think about it in the context of a board game where you place pegs to sink an opponent's ships:

By running only the regression tests (i.e., ○○) you're just repeating the same tests again and again. If a test discovers a problem, the system doesn't learn from it and alter the next test to run — it simply continues to run the next predefined test. This is very much like playing the game without getting any feedback from your opponent.

Using analytics and machine learning technology, we can analyze test results in real time and automatically learn from the results of previous tests to decide on the next action to take. Much like a human playing the board game would do when striking an opponent's ship — once they find something, they hunt around to find the ship they hit.

Learning from the results and feeding that data back to select (or in the case of Eggplant AI, actually generate) the next test case is key to optimizing test execution resources. Rather than predefine the possible list of test cases upfront and execute them sequentially, Eggplant AI interactively defines each test case as it's executed, and with each new step, learns from the previous runs. Eggplant AI makes this happen through two techniques: defining a model of your application, and using algorithms and machine learning techniques to build test cases.

**A Model of Your Application**

Eggplant AI is driven via a graphical tool to define a model of your application that simply describes the type of activities a user can undertake at any one time (or the API calls that can be invoked, or the mix of user interactions and API calls).

The model neatly defines the full set of possible test cases that can be executed and contains just two major elements.
**Actions**: The interaction a user can have with the system (or the API that can be called). For example, **click on the go to basket button**. Actions within Eggplant AI invoke an extensible set of test snippets to perform both the action and potentially test for success, and can use any technology — from image analysis (identify an icon), OCR (recognize some text), API (Invoke a RESTful or Selenium call), or any system-level call (invoke my bespoke C# application or shell script). For example, one text snippet may be: **Click on the basket icon and on the new screen we move to look for the my basket text**. If that completes successfully, then the next action is decided upon and the test continues.

**States**: Group actions together. For example, the **account** screen will include actions to change your name, address, payment details, etc.

In addition to the definition of user flows, each action and state can use data values and attributes.

**Variables**: Data fields that should be used within the tests. These can be defined (or generated) within Eggplant AI, or be taken from external data sets.

**Tags**: A data dictionary of any attributes and values. These can be defined and you can assign any of these values to any state or action. Tags are used to provide additional context for the bug-hunting system and to augment test scripts (logs) with specific values for more detailed, offline analysis.

Unlike a flow-chart-based system that would explicitly map actions to other actions (resulting in an explicit path with limited options, for example, **click on account and then click on username**), Eggplant AI does not define explicit paths between Actions, merely transitions between States. For instance, **click on account and then go to the account screen** means that after the account button is pressed, then all the possible options defined within the account screen may be used, not just the username action (that said, you can narrow down the options and limit flows in the model).

Additionally, Eggplant AI features Global Actions, which mimic the actions that a human (or API) would take outside the normal, expected flow. For example, if we are testing a mobile device, the user can rotate that device at any time. If we want to test for that in the conventional way of manually writing test cases, then where do we put the rotate command? It would be an arbitrary decision and only (and always) called at the specific stage as defined in the test. Within Eggplant AI, you can quickly define and invoke Global Actions at any point within any test case, and truly imitate the behavior of unexpected user (or API) behavior. Common Global Actions also include **Back/Forward** (pressing back then forward in the browser to see if any form details were lost) and **Delay** (stop interacting with the system for a period of time to simulate a distracted user).

Below is an example of an Eggplant AI model and the associated screenshots of the system we are testing: Note the **Rotate Screen** Global Action that can be triggered at any time.
Advanced Machine Learning Technologies

A model with multiple states, actions and variables that is simple to build can define billions of possible test cases quickly and easily. Because people can repeat tasks indefinitely, the theoretical number of test cases is infinite. But, we can't execute that many tests, and if we could, where do we start?

The answer is to use an ensemble of different models and algorithms, each of which focuses on a different aspect of what to do next. But just as important as the individual techniques is the final classifier that makes the ultimate selection of what action and data values to execute next. From the available inputs, the final classifier must decide the action to take at every step. For example, our Coverage algorithm may want to execute the Hotels action next, whereas the Bug Hunting model may want to visit Restaurants. It's the responsibility of the final classifier to determine what the best next step should be. In simple terms, it does this by comparing the confidence of each of its inputs, and then compares these relatively (in combination with any similarity in results) to make its selection.

One of the inputs is User Weights, which allows the tester (the person creating the model) to add explicit prioritization into the model. For example, if we want to explicitly bias testing Restaurants more than Hotels, we would set the weight of the Restaurants action to be greater than the Hotels action — increasing the likelihood of the Restaurants path being taken at any one time. However, the more important inputs come from the Coverage Analysis and Bug Hunting systems.

Coverage Analysis: This system will prioritize coverage across the model and try and direct the test flows (and data values) to maximize coverage. There are a number of different coverage algorithms used internally, and these are described below, however it is the amalgamation of these into a holistic model that is important to get a true sense of coverage. Within Eggplant AI, this is implemented in a proprietary way, which is not detailed in this document.

All Nodes (1): This is simply tracking that the element (an Action, a State, a Variable value, etc.) has been tested in any context at any time and within any flow or test case. Many systems use this mechanism to show an overall test coverage percent value, which is generally misleading because many actions are taken in context (for example, entering text into a field that is empty and entering text into a field that has text in it already), and so a measure of 100 percent in this metric omits any sense of context in testing.

All Pairs (2): This looks to cover pairs of paths between actions as well as any variable values used. All Pairs provides a measure of directly connected contexts (such as entering text into a field that is empty and entering text into a field that has text in it already), though with cyclic paths, you must take care to ensure that nonsensical and duplicate paths are not significant.

Extended (3) and Full Exploratory (4): Also referred to as 3rd and 4th order coverage models respectively, these are an extension of the All Pairs model in that they consider potential paths made up of combinations of 3 and 4 actions, and assess overall coverage based on how many of these have been completed. For complex applications, the number of potential paths based on these models will become extremely large, highlighting the importance of intelligent selection of testing patterns through mechanisms such as the Bug Hunting algorithm.
Bug Hunting: This system will look for common failure patterns across tests and direct test cases to prioritize paths that will actively detect bugs within the system under test. Bug Hunting is a sophisticated system that utilizes all the available context of each test, including the current test flow containing Actions and States, the set of variables and values used, the tags defined, etc. With the relevant data set, a human observer may detect a correlation of a small number of related factors if they’re sufficiently obvious (maybe one or two features, for instance, failures always occur on the iPad in vertical orientation). But the power of machine learning technology is that correlations can be detected across any number of features just as easily — for example detecting bugs occurring in our system when pop-up dialogs written in Angular-JS that contain text fields are used on iPads in vertical orientation. The process for this system at a high level is:

- When a failed test is detected, then the details of that failed test are passed into the system.
- All the attributes of the failed test are analyzed and correlations between failed tests are strengthened.
- This results in a set of weights or priorities that the Bug Hunting algorithm will associate with States, Actions, and Variable values.

To illustrate an example, let’s say we’ve defined some simple Boolean tags that we associate with our model, such as: Does the State have a text field? Does it contain a map? Is it considered to be a complex UI element? Now, once we run our model, the relative priorities of each of these tags will increase and decrease as tests pass and fail. In the example below, our first test failure is on the Restaurants action, which is associated with the map and text field tab, so our priority for those tags increases. Then, a separate failure at the search action increases the weight of the text field tag. This continues and the strength of the weights change until we get to the lowermost row below; here, based on the failure patterns we detected, those fields with both text fields and maps will be the ones we are most likely to target (highlighted in the bottom right image).
Obviously, this is a simplified example used to illustrate the principle and abstracts much of the underlying detail. It uses primitive Boolean tags and does not show the use of variables or other attributes. It also oversimplifies the weights and correlation calculations (read more details about this in the implementation section). For example, it's not a simple, unrelated correlation of data that is used as suggested above; not only do we consider the map and text field in the first row, but both the map and text field.

**Implementation Architecture and Techniques**

There are a significant number of proprietary techniques and technologies at the heart of Eggplant AI, and while we can't publicly disclose these for commercial reasons, this document aims to provide details of the overall architecture and approaches.

There are three logical elements within the Eggplant AI server:

- Orchestration Engine
- Neural Engine
- Data Layer

As the heart of the product, the Orchestration Engine includes the web server technologies for communicating with the client-side tooling and interfaces with the test execution engines to drive your application. The Orchestration Engine also hosts a number of statistical-based algorithms for results aggregation. While the complex analytics commonly fit into the Neural Engine, we also combine the results of those deeper analytics into statistical frameworks that reside within the Orchestration Engine.

The Neural Engine hosts the dedicated machine learning algorithms and models — the largest component is TensorFlow. To illustrate how we implement the Bug Hunting algorithm, we use a neural net with fully connected pairs and a final classifier based on a non-regularized Beta function. The size of the net automatically increases and decreases in size to accommodate the attributes (features) we feed it from the model; this allows us to scale dynamically to match complexity with the available hardware resources (i.e., processing time and memory).

In the Bug Hunting scenario, when a failed test is detected, the attributes of that orientation test are fed into the neural net implementation. However, we also feed in a recent test case that passed (i.e., without a test failure). This balanced pair implementation means that we do two things: Insert the set of attributes associated with a failure for the neural net to analyze, and pass through a completed test so that it also knows what passes and, critically, can unlearn. If we detect an issue with a specific type of control widget, for example, but then that problem is addressed by the development (or de-prioritize) that problem if it's no longer a problem. By also adding the completed tests that pass, we ensure that
Eggplant AI can unlearn fixed problems to target new defects instead. The neural net implementation utilizes Adam Optimization as both the Momentum and Exponentially Weighted Moving Average (EWMA) calculations allow us to avoid focusing on local minima (i.e., prevents us from over-focusing on small problems and helps us always focus on the main issues).

The Data Layer is where the status (i.e., prioritization) of each of the competing models and algorithms is stored. Decoupling the Orchestration Engine from the Neural Engine allows the test execution to be unencumbered by any spikes in computation required by the Neural Engine — it also easily supports cross-host scalability.

Productivity Benefits

This document outlines the purpose and high-level implementation details of Eggplant AI. It seeks to illustrate the benefits of using technology to efficiently generate test cases that both significantly reduce the manual effort required to produce them and make much more efficient use of hardware by always selecting the most effective test case to run next.

On the left, the figure below illustrates using an existing team to manually build out a set of use cases. The right side of the figure shows using that same team to build an Eggplant AI model that encapsulates all the necessary test cases in a very short period of time and then moving across to a different application. This results in huge productivity gains with existing resources in terms of application coverage and test-execution efficiency.

Eggplant Digital Automation Intelligence Suite

On its own, Eggplant AI is instrumental in helping organizations expand automation beyond test execution. Not only by automating test case creation from a model of the application but by using machine learning algorithms to select the best paths in order to optimize tests for coverage and find defects.

By pairing Eggplant AI with the rest of our products, you gain an intelligent suite of integrated solutions to power a user-centric approach to testing. So you can shift testing from a compliance activity to a revenue-generating profit center that shrinks time to market and drives customer satisfaction, conversion, adoption, and retention. All key elements that testing will be held accountable for in the digital world.

For more details about Eggplant AI, please send an email to sales@eggplant.io

Or, contact us in the USA +1 720 890 0211 / UK +44 20 7002 7888

About Eggplant

Eggplant provides user-centric, Digital Automation Intelligence solutions that enhance the quality and performance of the digital experience. Only Eggplant enables organizations to test, monitor, analyze, and report on the quality and responsiveness of software applications across different interfaces, platforms, browsers, and devices, including mobile, IoT, desktop, and mainframe. Learn more at eggplant.io.