AI as Co-Creator of Test Design

Excerpt from Sogeti 2021-22 State of AI Applied to Quality Engineering

eggplant
Test Automation Software

sogeti
Part of Capgemini

AI can augment the software testing process to help teams better understand the quality of their software, ultimately delivering better user experiences and more robust software.

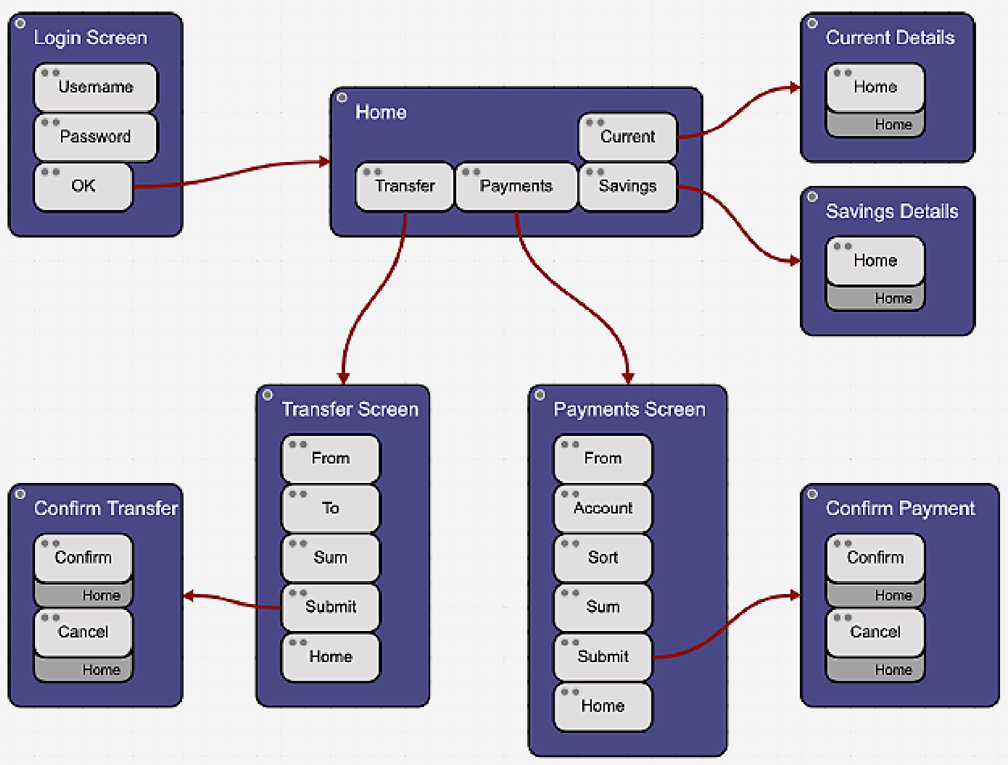
In the previous chapter, we met a close cousin to the model-based approach to testing, the digital twin. A digital twin is a virtual model of a process, product, or service. Digital twins can be used to understand the impact on real-world systems of a variety of inputs and scenarios. When we apply the concept of the digital twin to software testing, it means that the model can take into account a wider set of factors.

Traditionally, testers have created scripts that describe a 'happy path,' an ideal user journey that encompasses key actions in an application. However, to achieve a more systematic approach to testing, teams have moved to a digital twin approach. This focuses less on defining individual test cases, and more on describing the system as a whole.

Copyright ownership maintained by Capgemini, 2021.

This paper is an excerpt from 2021/22 State of AI Applied to Quality Engineering Section 2: Design, Chapter 2.

KEYSIGHT
TECHNOLOGIES



A digital twin – simpler representation with implied flows among the nodes (actions) within each screen (state).

With a digital twin, the goal is not to explicitly encode all known test cases, but rather to be able to describe all aspects of the system (or systems) we want to test. For example, in a mobile banking app, if a user can select ‘payments,’ ‘transfers,’ or ‘balance’ at a given point, the digital twin does not need any ‘happy path’ flows to be defined. It’s only when one of these options is selected that the next set of options presents itself. To make a payment, the digital twin only needs to know the options ‘from,’ ‘to,’ and ‘amount,’ rather than specify the order these fields are filled in. This approach diverges from typical model-based automation, because the various permutations of inputs can be tested — much as they would be in real-world situations.

AI-Driven Techniques to Enhance Test Design

To see how AI can contribute to improved processes and better software, we need to look at how software test automation can be augmented through AI. Broadly speaking, the difference between AI and simple automation is that instead of relying on following explicitly programmed rules, AI solves problems based on its understanding of an environment. In the context of testing a piece of software, the environment is the application and its current state, and the AI's ability to understand the environment is based on a model of the application and an interpretation of the elements within. As we will see, AI algorithms can be very useful for understanding and testing an application.

For the purposes of this section, we define AI as a system that is able to gather information from an environment, determine an action to be taken, and learn from the results of taking that action. Just as Jeff Bezos describes how AI is "quietly but meaningfully improving core operations," AI can improve the efficiency and effectiveness of software testing. In practice, this is achieved using a range of techniques.

Well-defined scenarios can be mapped out using "IF" statements, but this has the danger of becoming unwieldy and impractical for the numerous unique test flows, data, and environmental variations seen in a test framework. There are more advanced statistical approaches (variance, probabilistic), traditional machine learning (ML), or the even more advanced deep learning to build a data model than can be queried in production to help decide on an appropriate action. Finally, techniques such as unsupervised and reinforcement learning inherently change internal representations as a result of the systems with which it interfaces.

The real value of these techniques is to have a material impact on improving the effectiveness of the tests they produce. The purpose of rigorously testing an application isn't to reach some arbitrary internal KPI; it's to make sure that real users, whether customers or employees, don't encounter errors that prevent them from achieving their goals. And because real users might not behave in a manner that the application's developers had in mind, it's important to take into account a wider set of user journeys.

AI algorithms are used to help determine whether users will be able to complete their intended task and achieve their goal. Two categories of algorithms in particular help accomplish this aim.

Algorithm 1: 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 (perhaps one or two features cause failures to 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.

Consider this scenario: a test of a website reveals a bug. In the iOS device's browser window, there is a text box and a radio button. The model has learned something new, and thus will increase the probability of running other tests that involve text boxes and radio buttons on iOS devices. If the model finds another bug with the same characteristics, a pattern begins to emerge, and the system will train its focus on similar areas until all adjacent bugs are found. In this instance, the AI is taking on the attributes of a skilled, intelligent manual tester, learning something new and adjusting subsequent paths taken through a system. These learnings can then be fed back to development teams for quick resolution.

Algorithm 2: Coverage Analysis

How can visibility into depth of coverage be improved most effectively? A digital twin equipped with an AI algorithm for coverage allows you to clearly understand the areas of an application that have and have not been tested. 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.

- **All Nodes (1):** This tracks the element (an action, a state, a variable value, etc.) that have 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.
- **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.

Choosing a combination of the 4 coverage models allows the operator to optimize for different levels of coverage. This could mean testing the broadest range of user journeys in full exploratory mode or prioritizing a smaller set of business-critical workflows with much greater rigor.

5 Outcomes Derived from These Techniques

Eggplant's AI-driven testing approach generates several benefits:

1. **Optimize the use of resources for testing to release software faster**
   - You can easily define your test window and the metrics you care about. This allows the AI to run the set of tests that best fits that test window and maximizes the likelihood of finding relevant issues in the areas of the application you care about.
   - Eggplant helped a leading healthcare software provider accelerate delivery times by 23% and meet the challenge of rapidly changing regulations. These were achieved by improving dev-to-test ratios and saving costs by leveraging existing business resources without having to hire expensive technical resources.

2. **Identify defects more quicky**
   - The algorithms will quickly and relentlessly 'zero-in' on previously hidden defects to pick them out prior to release and earlier in the development cycle.
   - Using the AI algorithms and screenshots of every state change means that is easy to visualize the exact actions that are causing an issue. This reduces the mean time to fix by 90%.
   - AI algorithms optimize for a frictionless end user experience, rather than arbitrarily running of thousands of test cases to hit a certain pass/fail metric.

3. **Continuous intelligent test automation**
   - Testing runs can be scheduled in parallel and at scale 24/7. This means that testing is a continuous process, rather than a discrete and separate part of the development lifecycle. Continuous testing enables continuous delivery of improvements to software.

4. **Reduced maintenance**

   - Tests can be auto-generated and maintained with self-healing capabilities, negating the need for manual intervention to do updates for every new regression.
   - Eggplant helped the world's leading retailer reduce the cost of testing its point-of-sale systems by 47% via Eggplant's Fusion Engine, which can automate any technology and any device.

5. **Increases end user satisfaction**

   - Software that has fewer bugs and is updated more frequently will naturally lead to a better user experience.
   - Eggplant helped a large US bank increase its AppStore rating from 3 Stars to 4.5 Stars by shortening release cycles by 52% and improving performance by 80% using AI-driven test automation.
   - Eggplant helped a leading Japanese telecommunications provider increase NPS by 45 points by applying AI and predictive user and release analytics.

Bringing It All Together

In practice, the AI used to intelligently auto-generate test cases for exploratory testing will combine these strategies into an "ensemble" that will work together to select the next best action to take, continually learning as it goes. These algorithms all relate to test case generation, but AI can also be used to identify elements on the screen or adapt to changes in the application as new versions are released.

Imagine an onscreen submit button that has been moved since the last release. Its color and shape have changed, and in addition to these physical characteristics, the underlying objected identifier has also changed. Traditional testing would call for a manual update to a potentially long list of test scripts.

This approach can be circumvented through clever use of the aforementioned algorithms and the addition of computer vision, which describes the ability to extract meaning and intent from visual elements, such as text, images, objects, and interfaces. New elements can be intelligently identified without user intervention by looking back at historical changes for that element; seeing what other elements are nearby and finding patterns; looking for labels with a similar name; or seeing which elements have similar object properties to the element that is no longer there. The script is therefore dynamically adjusted and updated, allowing the test execution to continue uninterrupted and unsupervised.

The use of AI to enhance our understanding of software unlocks many opportunities. By having a firmer grasp on exactly how an application behaves in a wide variety of settings, scenarios, and conditions, developers can focus their attention on what's most important: delivering satisfying user experiences.

About the author

Jaspar Casey

Jaspar Casey is a Product Marketing Manager for Eggplant at Keysight Technologies. He has spent a decade bringing new ideas and digital products to market, covering everything from big data to blockchain. His work involves communicating the unique business value of AI-driven test automation.

Learn more at: www.keysight.com/find/eggplant

For more information on Keysight Eggplant products and solutions, please contact us.
Learn more about Keysight Technologies at www.keysight.com

KEYSIGHT TECHNOLOGIES

This information is subject to change without notice. © Keysight Technologies, 2021, Published in USA, November 9, 2021, 7121-1180.EN