Is Your Automation Any Good? Finally, An Answer.

Wayne Roseberry

wayner@microsoft.com

Abstract

It is time for a monstrous change in how we treat test automation.
Test automation has always been difficult to justify. The effort, the cost, the analysis - is it worth it? Sure, you find bugs, but are they the important bugs? Many people see the time and expense and wonder if it should be spent elsewhere. Those days can end. Using data analysis and customer telemetry, it is completely feasible to assign direct value to how the test automation relates to the real business value of faster delivery of what the customer really needs with higher quality. The key is to turn the test automation activity into part of the product itself, and the tests into just one more user of your system - a user that YOU control. You can know, for certain, which test activities are important, which ones are helping you go faster and which ones are finding the issues the users will care about because you will think about the test results the same way you think about end users.
This presentation will describe how the Microsoft Office team bridged the gap and collapsed the silos between product and test automation. It will show how the telemetry signal and analysis features and services that are a regular part of the Office product were the critical component to finally solving the age old problem of tests completely disconnected from customer needs and business value. It will show how individual product teams within Office use insights gained from bridging that gap to identify real problems earlier and ship with confidence faster and more often.

Biography

Wayne Roseberry *is a Principal Software Engineer at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, and all versions of SharePoint. Previous to testing, Wayne also worked in Microsoft Product Support Services, assisting customers of Microsoft Office.*

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne writes, illustrates and self-publishes children's literature.

1 Introduction

Shipping faster and more often necessitates rich, in-depth telemetry. The necessity of the feedback loop, the need to understand the customer quickly and thoroughly drive the creation of telemetry collection and analysis capabilities in any modern software team. Inevitably we will turn these same capabilities toward product testing.

We have less time to test, which motivates toward as much testing and issue discovery as possible via automated tests. This need for more automation increases the need for higher quality automation that is better at discovering issues sooner and is more relevant to business and customer needs. The same product telemetry capabilities that allow us to understand and analyze customers can do the same for our automated tests. We can better align the testing activity to customer activity and improve our ability to detect issues and bugs much sooner and faster.

2 Aligning the Tests with The Customers

2.1 When is a test any good?

For sake of this paper, a test is any good when:

1. It helps us discover and understand issues important to our customers and the business
2. We understand how it relates to behaviors and needs important to the customer and business
3. We understand what it does, how and when to execute it

There may be other criteria for a test being good or not, but this paper focuses on these points above.

2.2 Tests vs. Customer Separation

Automated tests and customers have always been separate.

We find it difficult to know whether test behaviors and results are relevant to the customer experience. It is difficult to create tests which cover the same experience one expects of users. A test engineer often guesses the value of a sequence or set of test steps. Would a customer actually do this, or would the issues found matter to the customer? Would the customers frequently do something missing from the tests?

We also find it difficult to understand test behavior after execution when we examine results. Short of reading test source code or logs, test behavior largely remains opaque. What actually happens inside the product when a test runs may be complex, rich and variant. This makes it even more difficult to determine how a given test outcome relates to the customer experience.

We used to address this separation by consuming test engineer resources and time. Allocate someone to spelunking the test results, wading through the details, and doing the best they can to report the right bugs and ignore the others all while trying to keep the customer in mind. This extra effort fit in long schedules with ample amounts of time. That time doesn't exist when shipping quickly, so the gap must close.

2.3 Telemetry Signal Makes Tests & Customers Speak Same Language

Telemetry gives us a medium to compare humans and tests.

Human use the product to satisfy their own needs or interests. Product telemetry generates data we analyze to understand customer behavior patterns, issues, problems and trends. We can anticipate adoption rates, spot rising anomalies that identify failure, assess if new feature changes are popular, or

simply understand which behaviors are most common or important. This analysis is a specialized skill, but once we have learned it we can apply it to anything where the telemetry is collected.

If the product already has a rich telemetry signal in place that signal ought to be collected in both regular customer usage and automated test usage. This creates an important transformation in the way we can think about tests.

2.4 Tests Become Users You Control

When tests and customers both speak the language of telemetry we can classify them under the same category. We can think of telemetry as "a data signal produced by users" and have two types of users – humans and automated tests. Biological and synthetic. Users we do not control, and users we do control.

We can compare the two types of users to consider how to think about them:

Comparing Types of Users		
	Automated Tests	**Humans**
Behavior	You can control behavior	You cannot control behavior
Priority	Priority/Reality of behavior unknown	Priority of behavior known All behavior real

Figure 1 Comparing Human and Test Users

We control automated tests, but we are uncertain of the realism or priority of the test behavior. We cannot control humans, but we are certain every behavior we observe is real and can establish priority of the behavior using relatively simple analytical models.

We should use these differences to our advantage. There are human behaviors we can force to happen via automated tests as a means of discovering bugs and issues prior to release. There are automated test behaviors we can prioritize by comparing to the human behavior patterns. All of this becomes possible when we think of both tests and humans as the same class of entity because they speak the same language (telemetry).

3 Aligning Via Telemetry

There are technical requirements and best practices to consider when collecting product telemetry along with automated tests.

3.1 Use the Same Code, Channel and Tools

You want to take full advantage of every investment made in the existing product. You want the telemetry data to look identical whether it is a test or a customer (qualifications later on this point). You want people to use the same tools to analyze the data from the same locations. You want to have every investment in fixing and maintaining the telemetry code, pipeline and tools to accrue to the automation test effort. You do not want to compete with time or resources between test telemetry and product telemetry. With that in mind, follow these guidelines:

1. Product code, not some separate test only code, should emit the telemetry
2. Upload through the same pipeline/mechanism as would normally happen with customers. At least make sure whatever code executes the pipeline is identical and not a separate pipe or service
3. Persist to the same data store as customer telemetry

4. Query, reporting and analysis tools should be identical as per product telemetry

Hopefully your automation and product telemetry are already in a state where the above naturally happens. If not, it is worth the time and effort to change and get automation and customer telemetry in sync. Any deviation will create unnecessary friction.

3.2 Emit Test Specific Telemetry and Make Product Testing Aware

The real power of capturing product telemetry during test automation is the ability to correlate the telemetry data back to the test specific data. This means that there will be telemetry data specific to running a test. The product needs to change to recognize when it is running a test. For example, Office added a specific event to its set of telemetry events that carries data about the test. The following data is uploaded by the Office client applications at startup if an automated test is running:

- An identifier for the specific instance of the test running, and an identifier for the job it was running in
- An identifier for the specific test running
- A tag to indicate if the test was running in lab or on an engineer's own machine
- A tag identifying the type of test job that was running (Office has different test types for different purposes)

The above fields are meaningful to Office and would likely change based on how a given team manages and executes their automated tests. Most of the identifiers specified above can be joined to other systems that provide more context, such as who ran the job, what branch version it was run under, and other data that relates to Office's engineering and release processes.

Probably the most useful of the test specific data is the test identifier. This allows later analysis that can correlate product behavior captured in the telemetry to a given test or set of tests. This correlation is the basis for comparing user activity to test behavior, analyzing coverage, predicting failure rates, redundancy and ROI on execution and a variety of other useful topics.

4 Applying the Telemetry

Once the automation and the telemetry are aligned, there are many useful questions you can answer.

4.1 Do we have sufficient coverage to ship?

Office releases product to a progressively growing set of users, referred to as "rings." The rings start with the product team (called the "dogfood ring", as in "eat your own dogfood"), then all of Microsoft, followed by select external customers that agree to get non-final builds, and finally all users in market. Slow usage patterns in the earlier user rings present a key challenge. Waiting for users to generate sufficient command usage to pass release criteria can add several days to ring promotion.

Some engineers check the coverage with automation. If humans do not use some commands enough then release engineers check the automation telemetry see if the same commands are getting test coverage. This helps the team decide if the volume of failures collected from the telemetry provides sufficient information to inform the promotion decision. The screenshot below depicts an actual dashboard that reports different commands inside one of the client applications as observed during test automation. According to the engineering manager on the team using this dashboard:

It gives us confidence around audience promotion. If we don't have human coverage per feature (by threshold) then we fall back to automation because we can feel good about the automation as a signal of usage as well... Both human and automation coverage is critical."

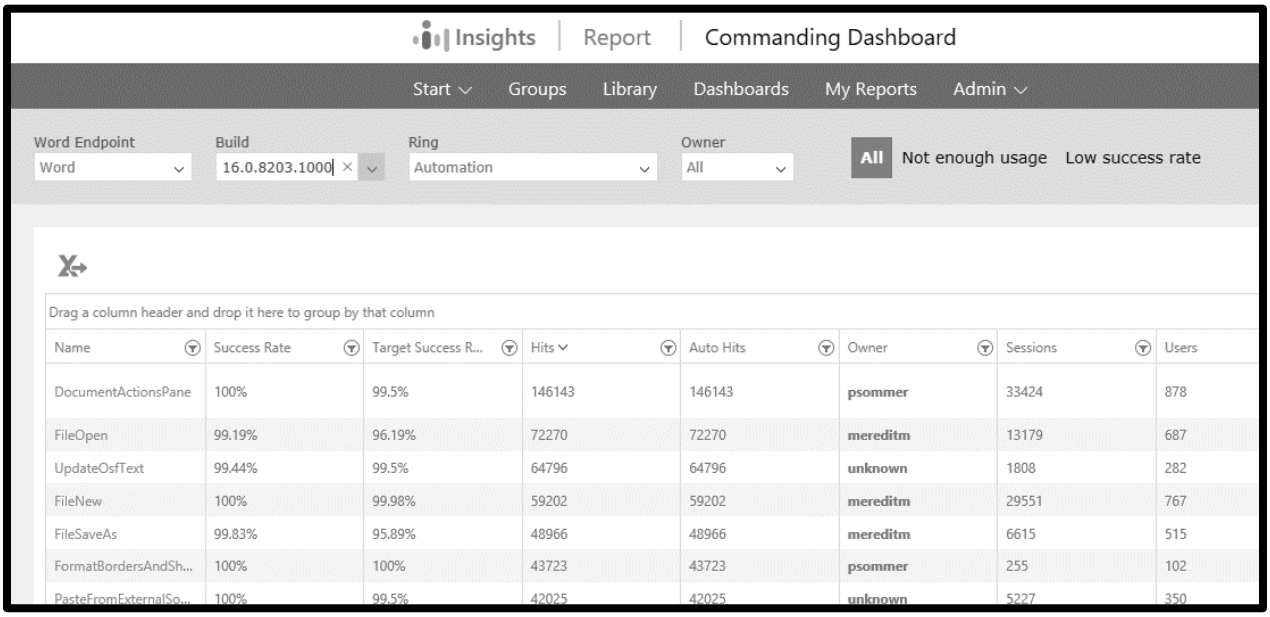

Figure 2 Command Usage Dashboard Depicting Automation Coverage

4.2 What did this test do?

Sometimes an engineer or a product team triaging test failure wants to know what a test did. Test logs only indicate test behavior from the test's point of view, they don't expose the inner behaviors of the product under test.

4.2.1 Test activity is no longer a mystery, the telemetry can describe the test

This is where product telemetry can help people understand what happened when a given test executed. The mystery is gone, as the events recorded in the telemetry will show how far the application got, what exact errors occurred when. Below is an example of a test result investigation page from the Office automation system's web UI showing events captured during a copy/paste test from one of the client applications:

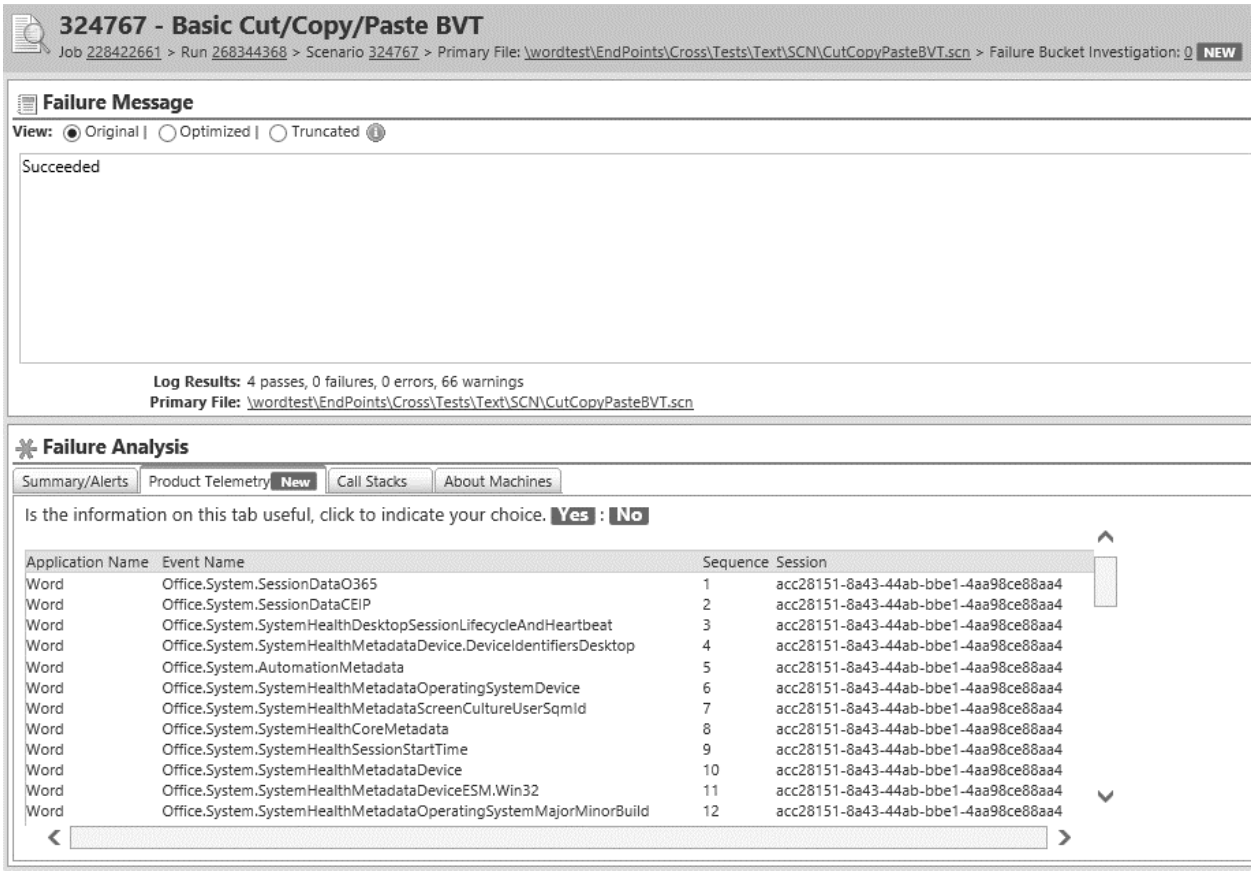

Figure 3 Automation Results UI Showing Telemetry Collected During Test

4.2.2 Makes variance more visible – not just pass/fail bug variance, but actual behavioral variance

Something that becomes apparent very quickly when comparing product telemetry across executions of the same test is that even when the test itself has no variance at all in its own behavior, the application underneath will vary a great deal in terms of exactly what events fire and in what order.

A six hour sample of all of the Office automation shows some interesting trends:

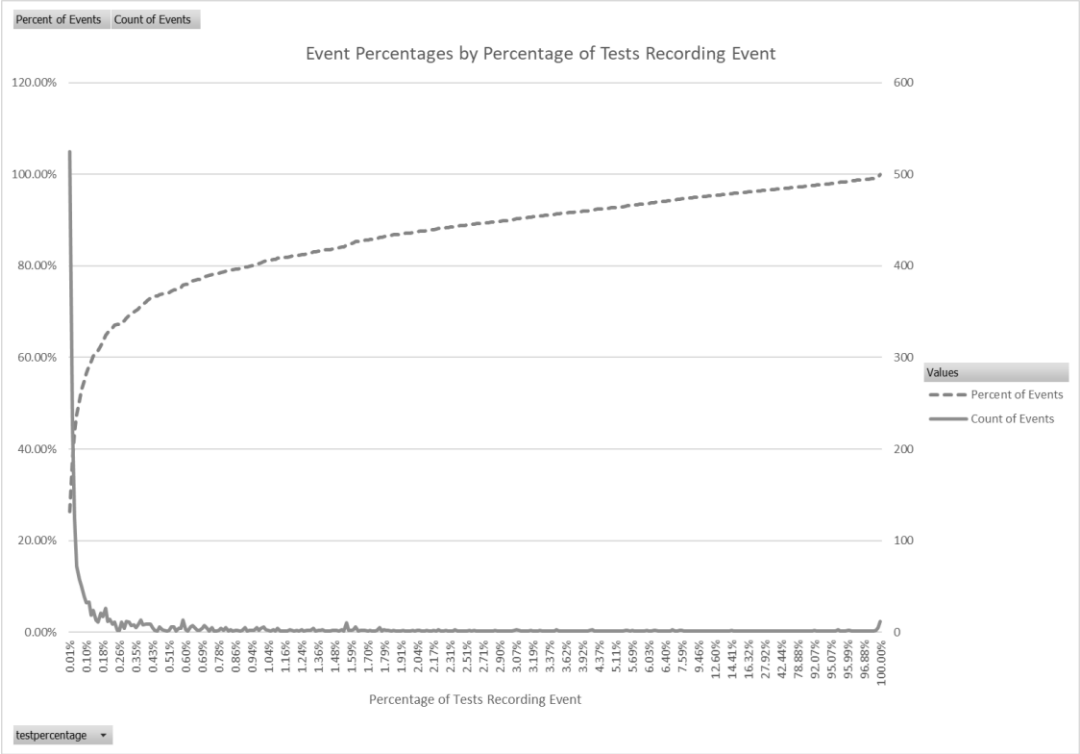

Figure 4 Event Distribution by Percent of Tests

The chart above shows the distribution of events by how many tests recorded them. Over half of the events affected only .06% of the tests, while approximately 3% of the events were recorded by 80% of the tests. This suggests a high degree of event differentiation between tests, but also that for any given test, at least 3% of the events recorded are not distinctive to that test.

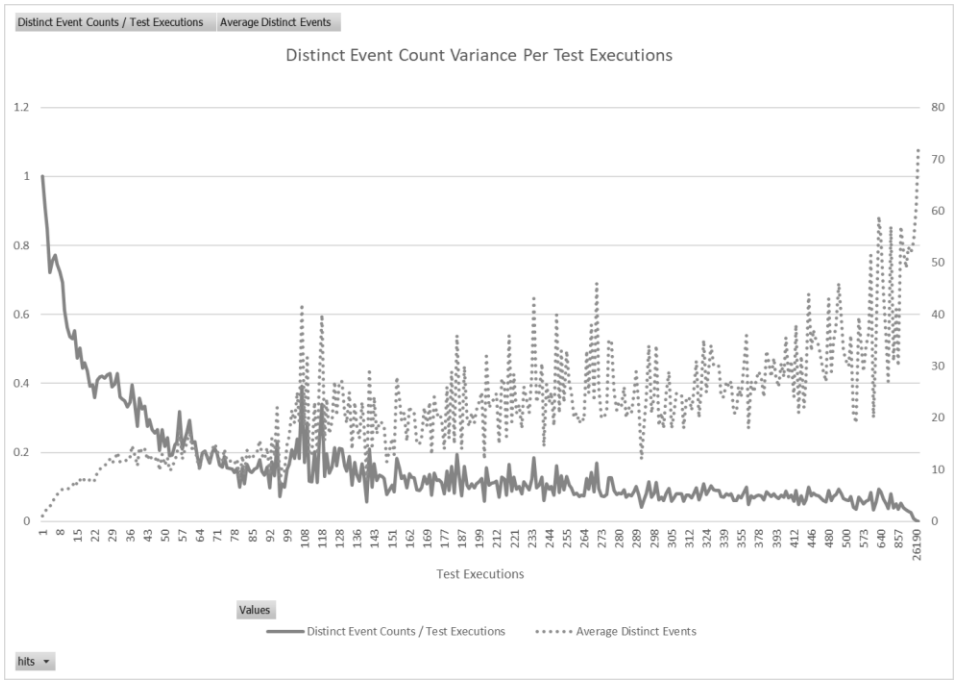

Figure 5 Event Count Variance Per Test

The chart above demonstrates the intrinsic variance in product behavior during test execution. For example, a test may record 30 distinct events during one execution, 31 in the next, 29 in the next and so on. As the number of test executions increase, the number of distinct events recorded per test trends toward 10% of the total number of executions. The trend continues upward, so long as the number of executions of the test increases. The interesting observation is that product behavioral variance in terms of combinations of events captured during a session continues as many times as a test is re-executed. Not shown, but event sequence variation trends to ~.99 of executions, so every time the test is executed, the sequence of events will be new.

4.3 Test Characteristics and Stopping Failures from Escaping to Customers

Once you can tie failures seen by tests to failures seen by customers and which tests suites the tests come from you can begin to understand the value of the different tests suites for serving different purposes.

In the table below "How Found" indicates different tests run by the Office team. "Escaped" means a human user later hit the a crash found by the test suite and "Stopped" means the crash was not seen later by human users.

How Found	Escaped	Stopped	Stop Rate	Notes
Developer Manual Tests	229	298	57%	No required stability or fix bar, no bugs ever filed
Product Team Full Suite	64	46	42%	No required stability or fix bar, bugs manually created
Product Team Check in Suite	11	25	69%	Checks quality of team branch, some teams auto-bug, usually gates main branch integration
Build Verification Suite Reliability Run	6	11	65%	Measures reliability of build verification tests, tests must be 98% reliable or disabled, bugs automatically filed
Dogfood Gating Verification Suite Reliability Run	2	1	67%	Measures reliability of dogfood gating suite, tests must be 99%, bugs filed on failures
Build Verification Suite	2	9	82%	Blocks release of build to product team, bugs automatically filed, failure locks team from checking integrating main branch
Dogfood Gating Suite	0	0	NA	Blocks promotion of build to dogfood users, tests must be 99% reliable or disabled, tests run twice to reduce intermittent failures blocking deployment

Table 1 Crash Discovery and Escape Rate by Test Suite Type

The tests in the suite increase in reliability as they are used for more and more stringent gating processes. As reliability and response to a failure increase, the likelihood a crashing bug will be found decreases. With this comes the likelihood that bugs found will be stopped before escaping to human users.

This is an important principle to recognize, and the data reinforces principles discussed in previous papers regarding managing test suite reliability (Roseberry, 2016). Test suites that are stabilized provide

reliable, fast, high precision information that can be used for managing process gates, but they lose the capacity to discover failures. As tests gain ability to find more failures, they also increase in noise rate and processes that normally ensure low escape rate (automatic bug filing, locking zones for changes, blocking build promotion, etc.) overwhelm team ability to respond. It is clear that a complementary set of test automation is necessary, rather than focusing exclusively on tests that are 100% reliable.

I am predicting that we will begin to see increased innovation over the next several years over precisely this dilemma. The pressure to ship faster, with a more certain signal will motivate us to apply better and better analysis to the noisy pile of less reliable tests to more quickly determine which failures merit our attention.

4.4 What tests hit this event?

Imagine you know which telemetry events to expect for a given user scenario. Or imagine you have changed a block of code, and you know which telemetry events ought to fire when that code is executed, or you know of several different ways the code is used and those ways are indicated by different patterns and markers in the product telemetry.

Do you have tests which cover each of those cases?

Product telemetry from test automation serves as a code coverage proxy. It won't get down to the block level accountability we are used to with most code coverage tools, but it will account for whatever level of telemetry instrumentation exists in the code in question. The screen image below depicts a query in Microsoft Insights and Analytics showing tests in the last day that invoked Microsoft Word's "ApplyHeading1" command.

```
let after=ago(1d);
Office_Word_Commanding_ApplyHeading1 |  where Event_ReceivedTime > after and Release_AudienceGroup == "Automation"
| project Session_Id, Event_Name | take 100
| join kind= inner (database("Office System").Office_System_AutomationMetadata | where Event_ReceivedTime > after
  | project Data_TestId , Data_JobId, Data_JobProperties , Session_Id ) on Session_Id
| summarize any(Data_JobProperties) by Event_Name, Data_TestId
```

Event_Name	Data_TestId	any_Data_JobProperties
Office.Word.Commanding.ApplyHeading1	206995	{ "JobID" : "235351779", "ClassificationID" : "1826", "ScenarioId" : "206995", "ScenarioName" : "NavPaneTabInsertAboveBelow", "ScenarioPath" : "OfficeVSO\OC\Word\Layout and Display\View\Navigation Pane",
Office.Word.Commanding.ApplyHeading1	223429	{ "JobID" : "235425620", "ClassificationID" : "-37", "ScenarioId" : "223429", "ScenarioName" : "NavPaneTabPromoteDemote", "ScenarioPath" : "OfficeVSO\OC\Word\Layout and Display\View\Navigation Pane",
Office.Word.Commanding.ApplyHeading1	274816	{ "JobID" : "235452512", "ClassificationID" : "1826", "ScenarioId" : "274816", "ScenarioName" : "NavPaneDragDropTabs", "ScenarioPath" : "OfficeVSO\OC\Word\Layout and Display\View\Navigation Pane",
Office.Word.Commanding.ApplyHeading1	206996	{ "JobID" : "235351779", "ClassificationID" : "1826", "ScenarioId" : "206996", "ScenarioName" : "NavPaneTabitemDelete", "ScenarioPath" : "OfficeVSO\OC\Word\Layout and Display\View\Navigation Pane",

Figure 6 Query Showing Which Tests Hit a Given Event

4.5 How do test users compare to human users?

One of the questions a product team wants to know is whether or not their tests actually cover what their users do. How much of a gap is there between the user experience and the test coverage?

4.5.1 Ring hit comparisons – developers vs automated tests vs humans in different rings

Comparing user behavior to automation coverage can be pretty straightforward once the telemetry is collected in the same place and is analyzed with the same tools. One need only count the distinct events collected on a per ring basis.

The following charts are examples of event comparisons between tests developers ran on their own desktop, tests run via automation in test labs and events collected via regular usage from human users of the product.

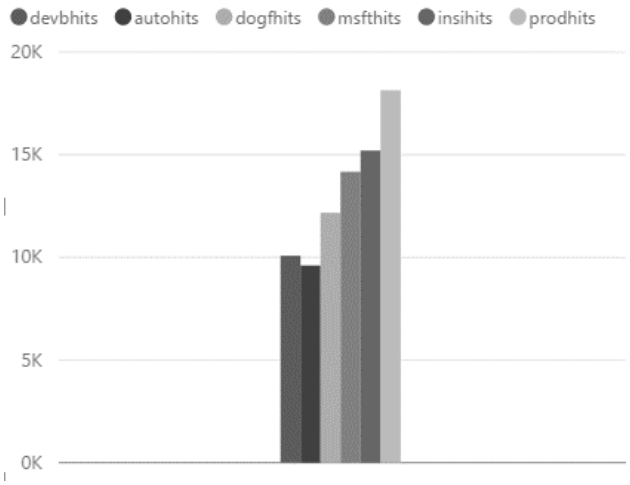

Figure 7 Distinct Event Count By Ring

The above chart demonstrates that production users (prodhits) generated nearly twice as many distinct telemetry events as seen by test automation (autohits). Meanwhile, developers performing their own tests generated slightly more events that test automation run in the lab (devbhits).

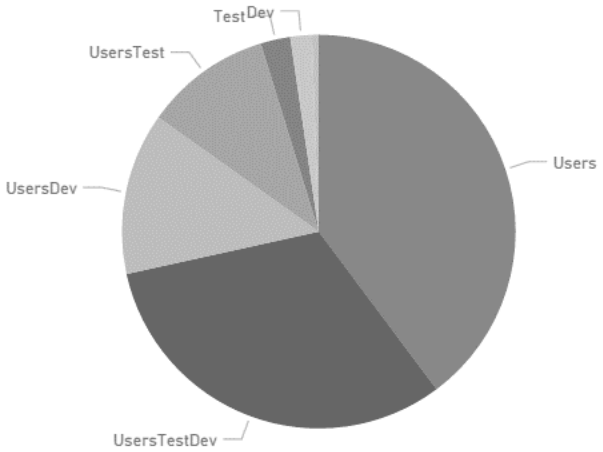

Figure 8 Event Overlap Based on Developers, Automation and Human Users

This chart compares event overlap. All the rings that comprise humans using the product (dogfood, Microsoft, insiders, Production) are collapsed into a single category called “Users.” Each wedge indicates the number of distinct events seen by each category. From here, there is nearly 40% of distinct events that only ever seen by human users. Meanwhile, about a third of events are seen by users, automated tests and developers. The remaining quarter is split between events tests hit that developers did not and vice versa.

One can use charts like the above to establish how large a gap exists between the test automation and different groups of users. One can also see how automated test activity and individual developer activity are distinguished by coverage.

4.5.2 Failure Hit Comparisons –crashes seen by humans versus by developers and tests

One can also compare coverage of specific failure types. For example, if the product telemetry system collects crash data, then one could compare which crashes were seen by automation or human users.

Office collects crash information from Windows. Each crash is assigned a unique hash tag based on the location of the crash inside the product source code. This has tag is persistent across builds of the product, so it is possible to track the historical record of a crash, but also to compare crashes seen in only internal builds versus crashes seen in released builds.

The following chart shows a crash that was detected during a product team private run of their entire automation suite (this is what “Comprehensive” refers to, versus “Comprehensive Official”, which is a run off the main trunk build across all Office check integrated changes). The crash was detected in automation one day prior to being seen by any human users, and two days prior to being seen in the centralized runs. In terms of process, this is interesting to the Office team, because the “Comprehensive Official” run has no gates assigned to failure, and the size of the suite is such that the official run occurs only once a week. So we have a case here were central runs straddled release of the code to dogfood, but the product team private run was able to see the crash first.

789f2482-cd6e-9117-0f6a-c7c616f6c480	30-Apr	16.0.8128.1000	Automation	Comprehensive	2
	1-May	16.0.8128.1000	Dogfood	Usage	4
	2-May	16.0.8128.1000	Automation	Comprehensive Official	2

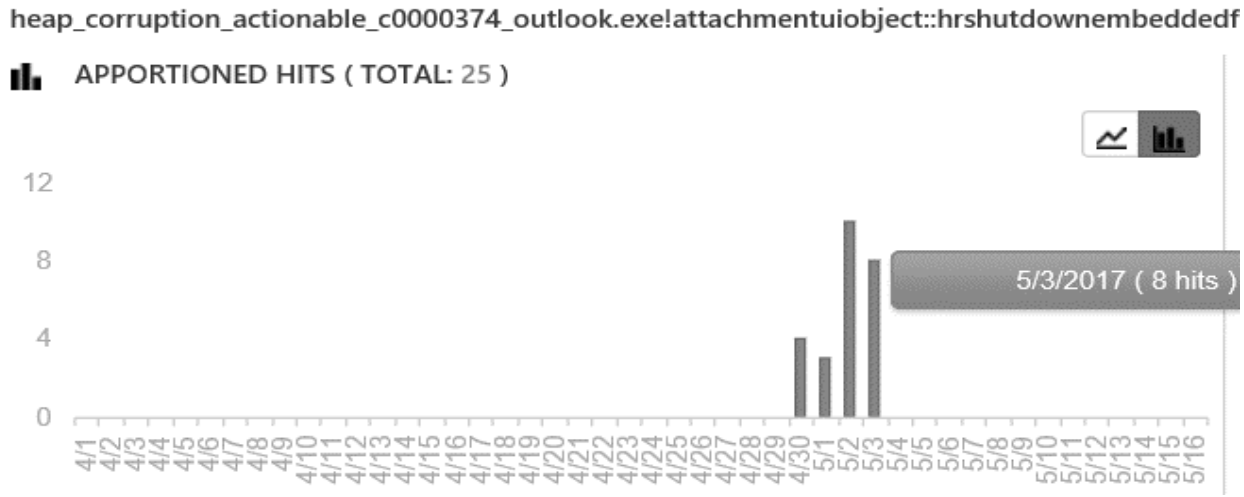

Figure 9 Crash seen in automation and dogfood

4.5.3 Anomaly Detection, Tests vs. Humans

A key benefit of using the same telemetry as the product is taking advantage of the analytical tools already developed for end users. The image below depicts an anomaly detection tool shows how activities within the Office client applications vary in behavior across different dates, builds, or groups of users (among other parameters). If certain attributes, such as activity failure rate or performance measurements fall outside expected ranges then they are highlighted as anomalous. Product team engineers create alerts on specific events of interest to them and if they feel a detected anomaly is a problem the tool will file the bug for them with all the relevant information. By framing “Automation” as a type of user, we can easily compare test automation against any other class of users and see if there are any issues we can spot in the differences.

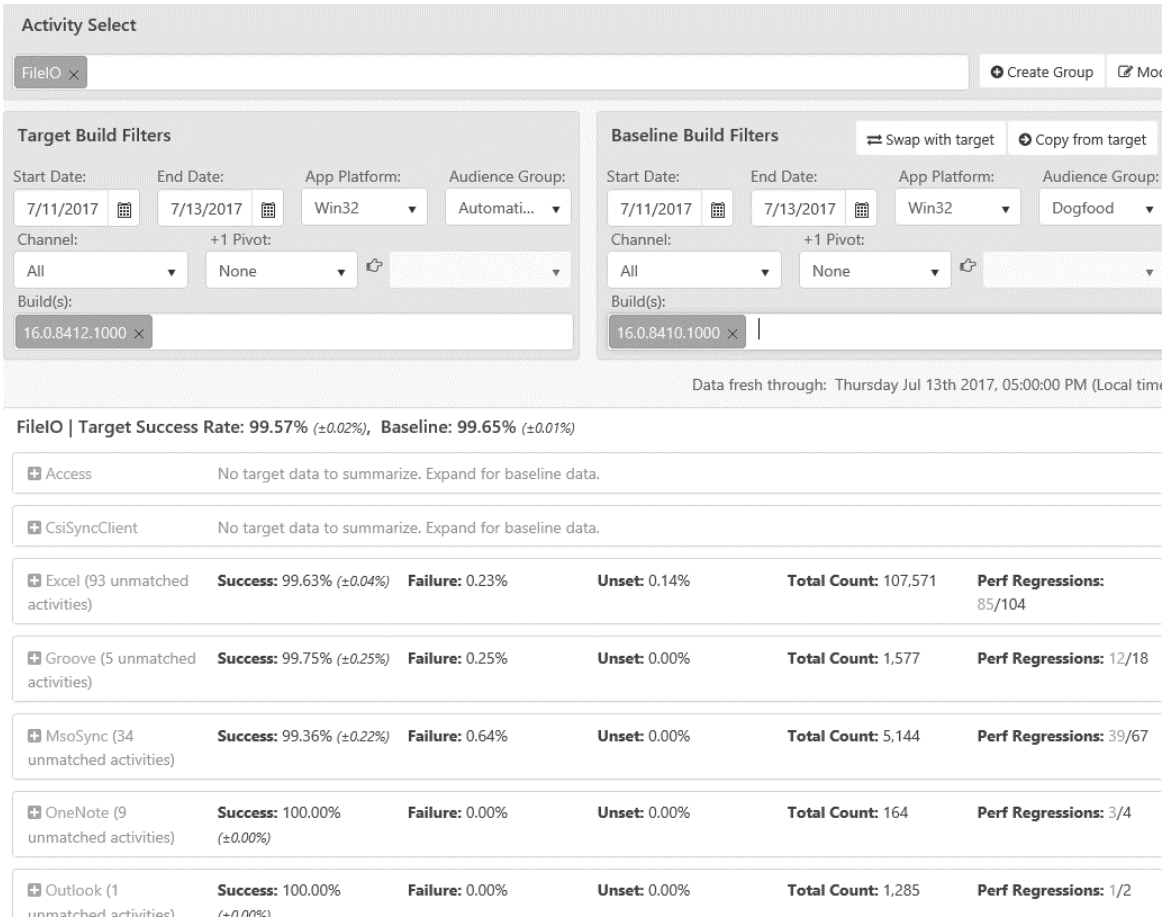

Figure 10 Anomaly Detection Automation vs. Customer Usage

4.6 Test Creation

4.6.1 See the Gap, Write the Test

One of the things people do when they compare event and telemetry coverage between real world usage and automated tests and see the gaps in the test automation coverage is plan authoring tests to address that gap. This is further assisted by measuring the occurrence rate of certain events, and addressing the most common events first. With deeper analysis of the customer experience, one ought to be able to

correlate certain events or event sequences with failure modes or decreasing customer satisfaction, which should also motivate more automation coverage against those behaviors.

This is a great low hanging fruit opportunity, and is usually easy to address right away. The task ought to reap early rewards, but it is also likely that there is a long “final mile” that becomes difficult to keep up with. Writing automation code manually is expensive, and chances are the uncovered events were uncovered for a reason. Teams tend to automate the easiest tasks first.

Also, bugs frequently exist not in the single action of one event at a time. The often exist in the combinations of different behaviors mixed together. Single, one at a time, automated tests do not handle this kind of approach well.

4.6.2 Markov Chain Models driving Automation

Another way to approach the test automation is to use the product telemetry to build a model that drives the testing behavior. A typical way of doing this approach is to sample telemetry sessions and build Markov Chain models of the frequency of transitions between different events. That model is then given to a test automation driver which executes single test actions as depicted by the model.

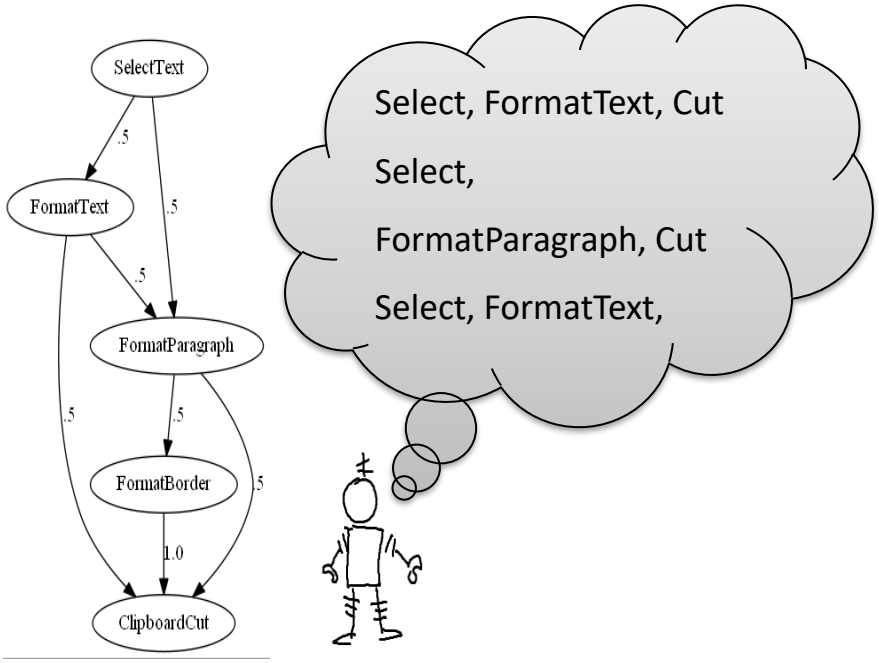

Figure 11 Comically amusing depiction of a Markov chain driven test

The approach is not like typical tests where each test is checking outcomes for a specific sequence. Instead, the test code is designed purely to drive the product behaviors. The “validation” happens by monitoring artifacts of product behavior, which can involve system event logs, memory monitoring, network monitoring, product logs and – of course – product telemetry. The exact sequence is more stochastic in nature. Not predictable or the same every time, but more seemingly random – but biased according to a model describing the frequency of user behaviors.

This is a common approach used often in server load, stress and capacity testing. In prior PNSQC sessions, I shared a methodology the Microsoft SharePoint team designed for building similar models based on web server logs (Roseberry, 2010).

In the past, such testing usually came at a high cost to set up a test bench, run and monitor the tests, and then spend several days mining all the likely sources of failure evidence to discover bugs. With the push

for faster releases, it is a good bet to predict a significant investment in innovation around automatically discovering failures from the telemetry and other artifacts. There is already investment in machine learning via anomaly detection and error prediction from real-world customer behaviors, so it is easy to imagine turning the same tools and techniques against synthetic load as well.

4.6.3 Teach the computer to test

Let's have fun and add an idea that is a little crazy, but not too far outside the realm of probable. At StarWest 2017, there is a session with the intriguing title "Machine Learning, Will it Take Over Testing?" (Merril). Indeed, will it?

On YouTube, there is an amusing video that demonstrates using genetic algorithms to teach a computer to play SuperMario. The key principle relevant to this topic is that prior to training, the machine had no understanding of the game SuperMario at all. All it did was emit numbers that were translated into movement commands, and based on that it was fed data that described the screen and score, although the machine did not understand it as such. The algorithm was trained over and over, optimizing on higher score values and keeping the game running (i.e. Mario not dying). Eventually, the machine learned how to win SuperMario.

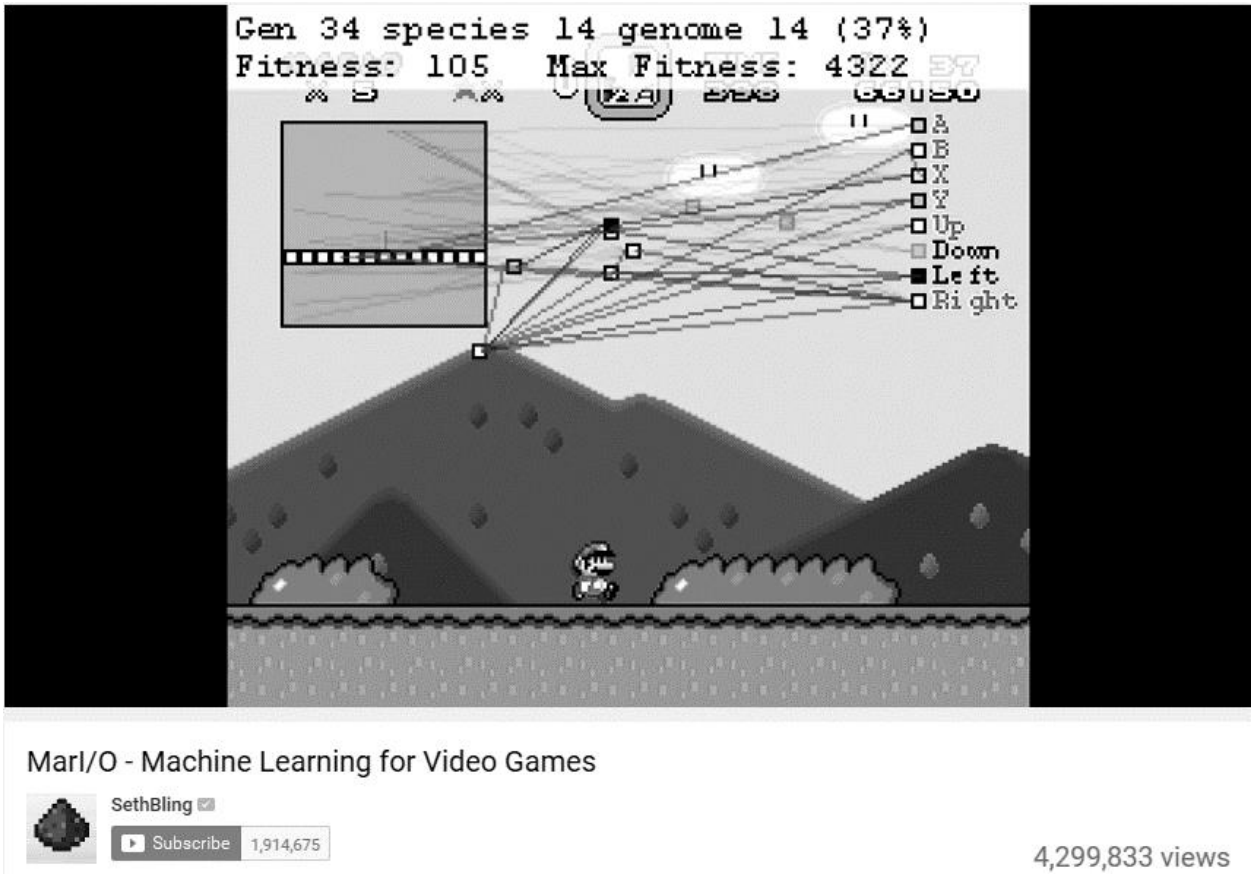

Figure 12 Screen shot of MarIO video

Let's extend this to testing. There are two things we can imagine: 1) training a machine to manipulate a product under test (click commands, type, etc.), 2) optimizing toward some test goal.

Clearly a typical software product is more complex than SuperMario, but in the same vein, the typical software team has more resources at their disposal than a home hobbyist playing with machine learning toolkits. It does not seem outside the realm of possibility that using a similar approach, a machine learning algorithm could teach itself to operate a given software product.

So now comes the question of what goal to seek. This is where the product telemetry is a useful input. Given an understanding of how telemetry manifests under test and real users, one could optimize for many different outcomes:

- Match user frequency patterns
- Maximize distinct event coverage
- Maximize distinct event sequences
- Optimize for higher failure rate per steps
- Maximize for distinct failures

The possibilities are broad, and it doesn't defy credibility too much to imagine the ability of such a system to vastly accelerate our ability to achieve much larger coverage, and hence failure discovery, in less time and with less human effort required.

5 Conclusion

Testing software is going to become more difficult as the demand for faster release velocity increases. It is time we apply the same discipline we have toward the software to the tests; better telemetry and deeper analytics applied to the actual product engineering and behavior. Once we align the automation with the product via telemetry the tests unify the gap between product and customer via the tests. This unification creates an opportunity for much better test quality and coverage than we have had before.

References

Paul Merrill, Starwest 2017, “Machine Learning, Will it Take Over Testing?” https://starwest.techwell.com/program/concurrent-sessions/machine-learning-will-it-take-over-testing-starwest-2017

SethBling, June 13, 2015, “MarI/O – Machine Learning for Video Games“, https://www.youtube.com/watch?v=qv6UVOQ0F44

Wayne Roseberry, PNSQC 2010, “Simulating Real-world Load Patterns When Playback Just Won’t Cut It”, https://www.pnsqc.org/simulating-real-world-load-patterns-playback-just-wont-cut/

Wayne Roseberry, PNSQC 2016, “Winning with Flaky Test Automation”, https://www.pnsqc.org/winning-flaky-test-automation/