A Distributed Randomization Approach to Functional Testing

Author(s)
Ilgin Akin & Sam Bedekar

Abstract

Traditionally, much of software testing focuses on controlled environments, predefined set of steps, and expected results. This approach works very well for tests that are designed to test targeted functionality of a product. However, even when test automation results are all green, have you wondered what other bugs might be hidden in the product? Have you hit a point in your test automation where you felt like you are not finding new bugs anymore but only regressions?

These are some of the questions we had been thinking about at Microsoft that led to the AutoBugbash project. A standard bugbash is a focused amount of time where the whole team gets together in a room and pounds on the product. Bugbashes are a highly effective way to find a lot of bugs. There are more eyes on the products and people tend to do a lot of random actions that may not be part of their day-to-day structured testing. However, bug bashes are expensive in terms of manual labor.

AutoBugbash is a form of Bugbash automation that incorporates the elements of decentralization and randomization. There are two main components to this approach. The first component is a set of standalone test clients that autonomously take actions on their own and verify expected behavior locally and log their observations with minimal state checking and no predefined script. The second component is a post-run component called the reconciler which reconstructs the sequence of events by parsing logs and matching events to actions. With this paper, we will describe how the AutoBugbash project helped uncover crashes and other hard to find bugs within the Microsoft Lync product.

Biography

Ilgin Akin has been working in the Unified Communications group at Microsoft as a Software Development Engineer in Test since 2006. Prior to Microsoft, she has worked at Pitney Bowes Corporation as a Software Developer.

Sam Bedekar is a Test Manager at Microsoft. He has worked in the Unified Communications space for nine years. Sam is very passionate about test and the impact it can have on product quality and the industry. Sam believes that Test Engineering has vast potential to grow and is excited about leading efforts to bring the state of the art forward.

Acknowledgements

Bo Yang, David Moy, Jack Banh, Matt Mikul

Background of this Case Study

Microsoft Lync 2010 is an Instant Messaging, Presence, VoIP, and Collaboration client that enables remote meetings & communication scenarios. In the context of testing, Lync provides its own set of unique challenges and environmental factors, such as:

1. Most scenarios are distributed across multiple client endpoints/machines.
2. Scenarios can be run on a desktop client, IP phone client, mobile client, or web client.
3. Environmental factors such as operating system and Microsoft Office version can affect the test results.
4. A test case can be written at the user interface (UI) layer, or directly on top of the object model layer.
5. The implementation of the business logic is asynchronous by nature – multiple operations/actions can be sent to the server with responses arriving later. A transaction (or scenario) may involve multiple clients to complete.

1.1 Sample Test Case

One typical scenario in Lync is establishing a call between two endpoints. The basic steps for this test case are:

1. Endpoint-A makes a call to Endpoint-B.
2. Endpoint-B accepts the call.
3. Verify the call is connected on Endpoint-A.
4. Verify the call is connected on Endpoint-B.

In this setting, three entities come from those steps: the controller (or driver), the Endpoint-A, and the Endpoint-B. In a typical implementation of this test case, each endpoint is running on its own machine, while the controller may be running on a separate machine, or on either endpoint machine. In both cases, the three entities communicate remotely. These entities are a common theme throughout our test portfolio.

1.2 Testing Meetings

To give additional context on meetings, a meeting is a virtual conference or meeting room.

1. The Lync meeting room will have several Modalities – Instant Messaging, Audio Conferencing, Video Conferencing, Whiteboard, Application Sharing, PowerPoint sharing, and Handouts.
2. The meeting will have several participants – each joined user is a participant
3. A user may join from multiple endpoints – for example, a user may join audio conferencing from their phone, and application sharing from their desktop PC.
4. Users fall into various permission categories – there is one organizer, multiple presenters, and multiple attendees. Each permission category determines which actions a user is allowed to perform.
5. Each participant, based on role, can interact with various Modalities.

1.3 Examining a Test Portfolio

To test Lync, we needed a diverse set of test cases comprising an overall test strategy. We call our set of test cases a *test portfolio*. A traditional test portfolio consists of test cases of various *test targets* and *test*

techniques. A test target is a strategy to test a particular aspect of a product. For example, functionality, performance, and stress are all test targets. A test technique is a method for testing a particular target. For example, functional tests and beta tests are both techniques that can be used to test functionality. A technique may be classified as automated or manual. An automated case has several advantages – low cost of execution, consistency in methodology. A manual test case has a different set of advantages – appropriate randomization, real-time verification, as well as advanced heuristics for verification, among other advantages.

1.4 A Sample Test Portfolio:

Test Target	Test Technique	Type
Functionality	1. Functional Automation	Automated
	1. Internal Usage of the Product	Manual
	2. Customer Beta Testing	Manual
Performance	1. Time Measurements in a Lab Controlled Environment 2. Time Measurements from user usage data	Automated & Manual
Stress/Scalability	1. Lab Environment stress 2. User Model based scalability	Automated

1.5 The "Bug Bash"

Over time, a technique of manual testing evolved that we called the *bug bash*. The idea was to have a set of folks come together to explore and find bugs in the product in a fixed set of time. The technique was manual in nature and focused on volume of bugs in a short burst. The directive is to "test at will." There are a few aspects that make a bug bash successful:

1. Real Time Decisions – As bug bashes are either unscripted or have broad generalized scripting, testers will often choose their actions. This adds an element of randomization when bringing several testers together.
2. Avoid real time verification that may slow down or change the scenario – Folks would tabulate issues and record bugs later.

This technique was very effective as it found bugs that spanned many test targets. Functionality bugs, performance issues, and stress related bugs are often exposed through bug bashes.
Over time, the bug bash became a staple of our test portfolio. It was a great supplement to many of our other test techniques. As it became a first-class citizen, we started discussing the possibilities of automating the bug bash. We realized the irony of attempting to automate a technique which brought benefits by being manual & randomized. We broke down the components of a bug bash and examined each one to see how we could automate it. Along with that, we had a few clear goals:

1. Repurpose existing automation – this was a clear goal to avoid duplication of effort.
2. Allow cross-platform test cases
3. Simulate a real "meetings" bug bash – allow each simulated user to perform their own actions
4. Test without modifying the meeting cadence – though this is test automation, it should simulate the speed and validation model of a manual bug bash

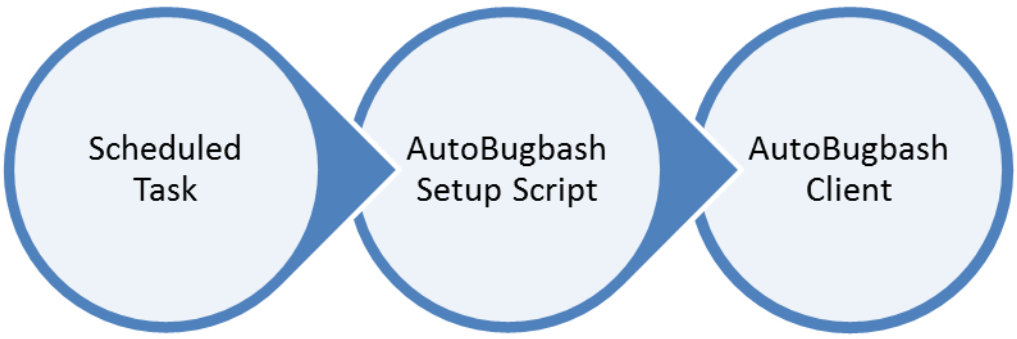
All of these became factors in our design for the meeting auto-bug bash. The meetings operating model was as follows:

1. Users can leave/join as they please – a full-featured meeting is full at 250 users.
2. Users can add a Modality which introduces the Modality into a meeting. Each participant can then choose whether they want to participate in the Modality or not.
3. Typically, when a user performs an action, such as sending an instant message or flipping a PowerPoint slide, other participants see the result of that action (receiving a message or viewing a slide flip respectively).
4. Several users can perform actions simultaneously

1. Architecture of AutoBugbash Framework

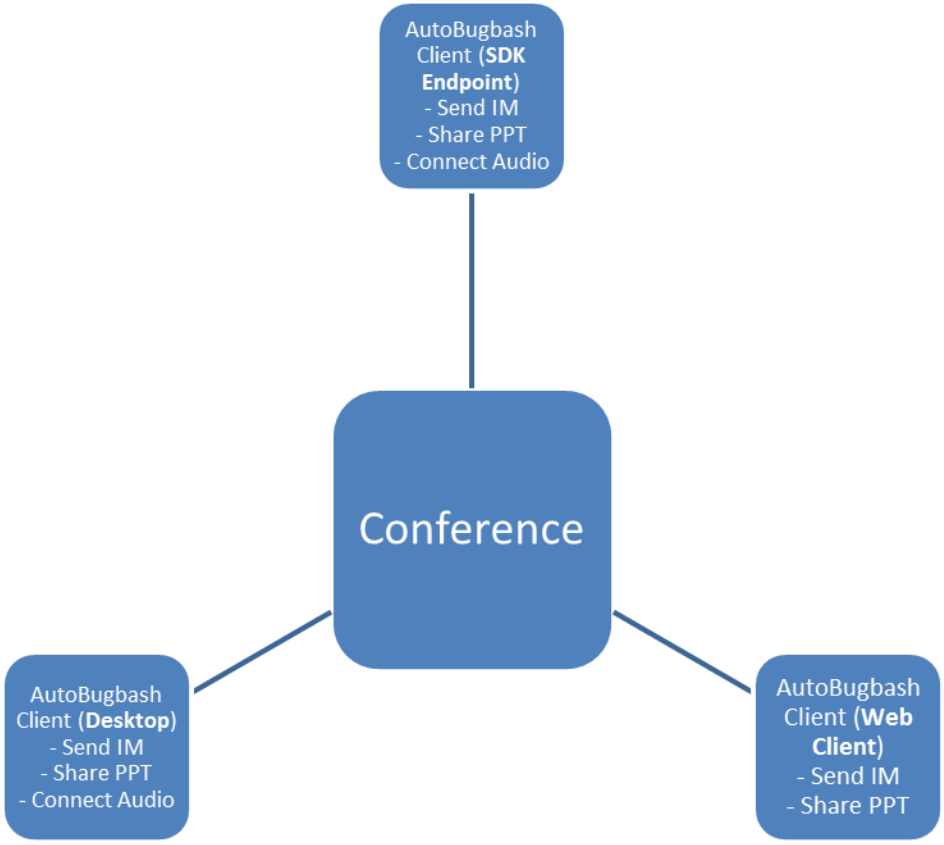
AutoBugbash framework consists of distributed AutoBugbash clients and a Lync meeting that these clients join and perform random actions within a specified amount of time. Examples of actions are sending instant messages, connecting and disconnecting audio, and real-time desktop sharing. During the run, AutoBugbash clients act autonomously and there is no centralized component to control and/or synchronize the clients. After the run is completed, each client uploads its logs and other files, such as dumps, to a shared log repository. Following this, a component we call the reconciler parses the logs, matches each client's actions to events and creates a timeline of the meeting from start to end. The output of the reconciler is a report file that details the run. There are three stages to the AutoBugbash framework.

2.1.1 Pre-Run



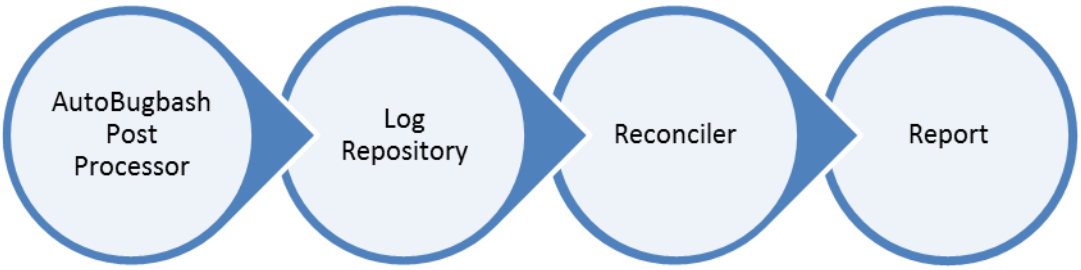
The AutoBugbash pre-run starts by a scheduled task that kicks off the AutoBugbash setup script. The scheduled task is setup on each machine that is going to run AutoBugbash clients. The setup script is responsible for determining the build to use, copying the necessary test and product binaries to the machine and starting up the AutoBugbash client.

2.1.2 Run



During the run, each AutoBugbash client joins a pre-assigned online meeting, and executes random actions until the end of the run. The clients do not wait for each other and do not communicate with each other through. One client might be sending an instant message, while another might be connecting to audio Modality, and another client sharing a presentation. This approach allows the run to be completely non-deterministic. The clients finish the run at a pre-determined time and exit.

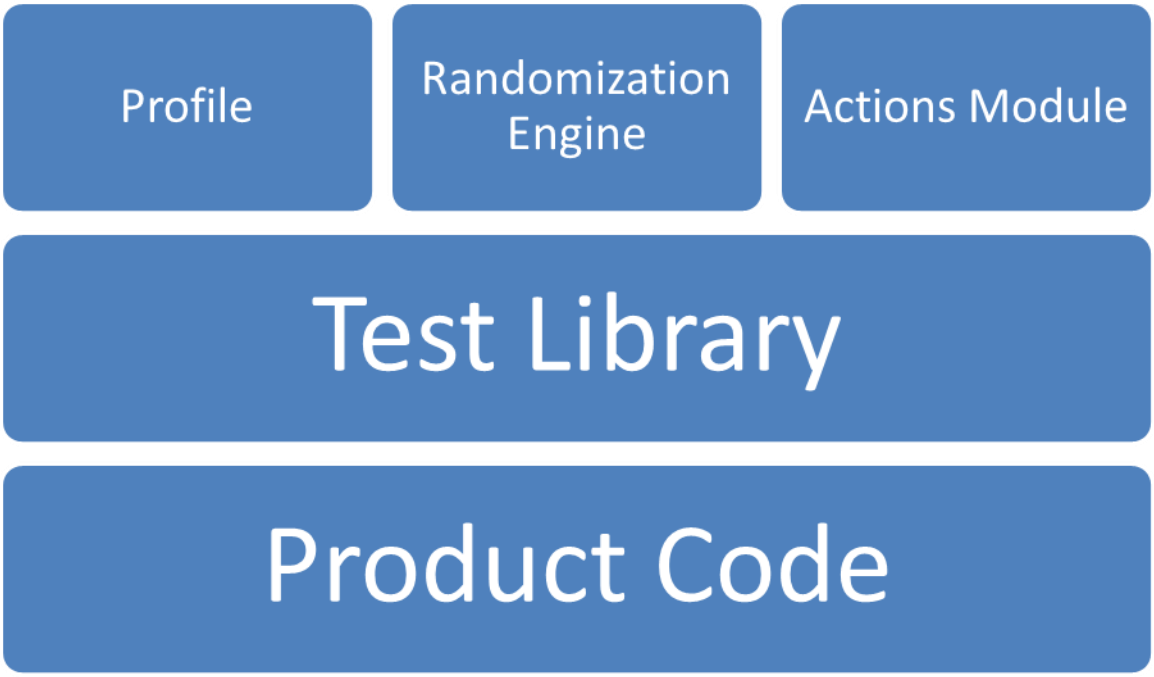
2.1.3 Post-Run



After an AutoBugbash client exits, a post processor script runs and uploads logs and dump files for the client to a central log repository. The log repository is a shared directory, and each client uploads to a separate subdirectory. Following this, the reconciler tool is executed and produces a report file of the run.

2.2 Architecture of AutoBugbash Client

2.2.1 AutoBugbash Client



An AutoBugbash client is a standalone automated test client with AutoBugbash specific components on top of the test library. The main components that sit on top of the test library are the *profile builder*, the *randomizer*, and the *actions* component. The added components are designed to support extensibility and reuse of existing automated test clients for the AutoBugbash framework. Extensibility of the AutoBugbash framework and of the client is designed at three levels. At the reconciler level, any client that adheres to the formatting and rules of the reconciler can plug into the framework.

At the Endpoint Factory level, any endpoint type can be plugged into the framework, as well as at the Action Factory level where multiple sets of actions are supported.
In the following sections, the components will be described in detail.

2.2.2 AutoBugbash Client Call Flow

AutoBugbash client's execution starts with building a profile which the client can follow. The profile uses the Endpoint Factory to create a specific endpoint type, following that a scenario for the AutoBugbash client is created. A scenario is mainly a list of actions that the client can execute based on the type of endpoint. Then the endpoint is initialized, and ready to execute. Throughout the run, the AutoBugbash client runs in a loop, picking a random action from the scenario, executes it and moves to the next action.

2.2.2.1 AutoBugbash Profile

Microsoft Lync supports multiple endpoint types such as UI, OCOM, OCOM.NET, mobile and web endpoints. A Profile describes a endpoint type and a scenario. A Scenario describes selected actions for the client to execute and how the actions are distributed. Each endpoint is capable of executing a subset of all actions. For example, while the Lync desktop client is capable of making calls, the web client is not capable of performing that action.
The Profile component reads an XML configuration file to understand which endpoints are capable of which actions. The XML configuration file also assigns weights to both endpoint types and actions associated with the endpoints. The profile component then selects a endpoint type using the weighted randomization algorithm, and prepares a scenario for the AutoBugbash client to run. There are actions that are not part of the profile, such as creating an online meeting, signing in, and out. Leaving these actions out allows us to have some control over the run, and allow the clients to do more actions. The way the profile picks an endpoint and prepares the scenario is through a weighted randomization algorithm. Each endpoint type and each action has a weight associated with it.
Being able to configure the weights through a centralized configuration file, it is possible to specify the focus of the run for that day. For example, to focus on testing endpoint type A, set the weight of endpoint type A to 80, and endpoint type B to 20, in the configuration file. This way, more clients will pick endpoint A.

```
<GlobalProfile>
 <Endpoints>
  <Endpoint>
   <Type>OCOM</Type>
   <Weight>80</Weight>
  </Endpoint>
  <Endpoint>
   <Type>OCOM.NET</Type>
   <Weight>20</Weight>
  </Endpoint>
  <Endpoints>
 </Endpoints>
</GlobalProfile>

<IndividualProfile>
  <ActionMix>
   <SupportedEndpoints>
    <Type>OCOM</Type>
   </SupportedEndpoints>
   <Actions>
       <Action>
 <Name>AudioConnectAction</Name>
    <Weight>200</Weight>
   </Action>
   <Action>
<Name>AudioDisconnectAction</Name>
     <Weight>200</Weight>
    </Action>
   </Actions>
  </ActionMix>
</IndividualProfile>
```

2.2.2.2 Randomizer

The Randomizer component is a simple component that makes use of the Action class. Any class that extends the Action can be passed to the Randomizer and can be randomized. The Randomizer component provides both standard and weighted randomization. It can both generate a scenario which is a list of randomized actions, and it can also return a random action from a given scenario.

2.2.2.3 Actions Module

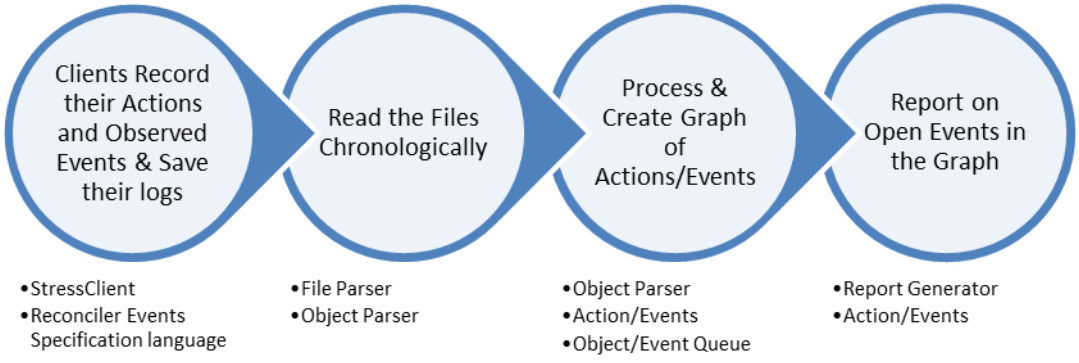
The Actions module is a layer that is built on top of existing test libraries. This module standardizes actions that can be executed by the AutoBugbash framework, allowing any type of endpoint to plug in and have the AutoBugbash framework execute its actions. The Actions module also keeps endpoint specific code separate from the randomizer component.
The Actions module contains actions for endpoints that extend the Action class. The Actions module follows the factory pattern, where it provides the specific Actions factory for a given endpoint type. Then the specific type of endpoint's action factory provides the specific action based on given parameters.
This Action class defines a method called Execute, which accepts an optional dictionary of parameters. The Action class extends each action type, supported by the particular type of endpoint. In the implementation of the Execute method, the applicable test library methods are called. The guideline for Action implementations is to do minimal state and event checking for each action. The state checking only checks whether the action can be invoked. There is also minimal event checking after the action is executed. This lessens the total time spent executing an action and allows more actions to be called in sequence. It also allows more code paths to be executed within the product code.

2.3 The Reconciler

One of the key tenets of the auto-bug bash is to do post-mortem analysis of expected functionality rather than real-time verification. This has several advantages:

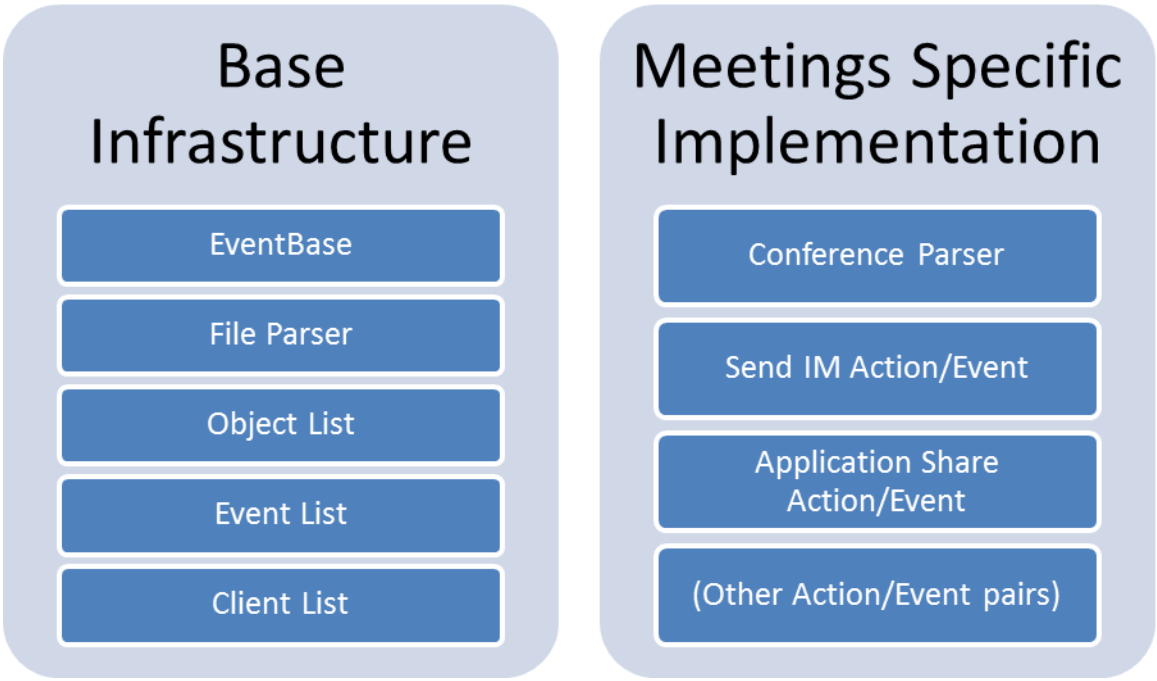
1. Ensures that verification does not slow down or change the scenario being tested – if 300+ clients were doing real-time verification when an action was taken, there would be significant slowdown in the execution.
2. Allows simple scalability – without a centralized component to coordinate actions, it is easy to remove or add clients at any time.
3. Allows multi-platform support – Doing post-mortem analysis allows there to be clients of various platforms joined to the meeting. A central component does not need to be able to speak to clients of multiple platforms (for example, mobile platforms, C/C++, C#, Silverlight, Java, etc.)

2.3.1 Reconciler Workflow



Clients will record their actions and events into a common format we call the **Reconciler Events Specification**. Regardless of the platform, a client can record its actions and events in this format for the Reconciler to parse.

2.3.2 Reconciler Components



2.3.3 Endpoint & Endpoint List

- Action Capabilities – lets the reconciler know which Action/Event pairs a client supports
- Client URI (Uniform Resource Identifier) – a string identifier identifying a client (such as user1@contoso.com).

2.3.4 Action/Event

The Reconciler breaks down the meeting logs into Actions & Events. An Action creates an expected Event in an "open" state. As the Reconciler continues to parse the log and encounters the Event, the open Event object is updated to reflect the clients that received that event.

- Event State
- Reference Count by participant URI
- EventBase class which provides basic matching infrastructure

Example:
The SendInstantMessage action creates an InstantMessageReceived event in an open state with each participant with a reference count at 1. The event is added to the event queue. As each participant receives the event, that participant's reference count goes to zero. When the last participant receives the event, all participants' reference counts are zero, and the event is marked as "closed."

2.3.5 Event Queue

The Event Queue is a list which keeps track of all created events. The reporting mechanism reviews all events and determines which events are still open.

2.3.6 Object Parsers

Object Parsers are parsers that understand a specific set of actions & events and act as factory methods for Event objects.

2.3.7 File Parser

The file parser reads all log files, processes them chronologically, and sends each corresponding line to the Object Parsers.

2.3.8 Meetings Specific Implementation

In the meetings-specific implementation, the Conference Parser object keeps track of Participants & Modalities in the conference. It creates Modality-specific events. Over time, the conference parser recreates the conferencing scenario. If an instant message is sent but not received by all capable endpoints, that event remains open. An open event is the indication of a functional failure; that is that all expected consequences of an action (in the form of events) did not occur as expected.
When the reporting processing is happening, all open messages are flagged as failures in the final report.

2. Rhythm of Execution

To get the most benefit from AutoBugbash, the best approach is to deploy the framework to a high number of distributed clients and have a long span of time for the run. Since many bugs are found on real user machines, we decided to deploy AutoBugbash on our team members' machines. To not interfere with their daily work, we provided a scheduled task to kick off an AutoBugbash client per machine. The run would start at 12am, and stop at 7am. This approach allowed us to use a pool of machines without disruption to the users.
We also automated the process of getting logs and dump files, in case of crashes, automatically. Selected members of the AutoBugbash team can look at dump files, logs and report from previous night's run from a central location and triage the issues.

3.1 Results

With the auto-bug bash, we hit several classes of issues:

1. Memory Leaks
2. Timing issues – Crashes, Incorrect Functionality, Incorrect state for the product
3. Reliability Issues -- Dropped Messages

3.1.1 Sequencing & Conditions

We did not hit many of these issues during functional tests. We found that running any small sequence of these steps in isolation would not reproduce the problem. But the exact conditions created by the constructed scenario would be different over time & expose different issues that only occurred under those conditions.
For example, switching sharers in an application-sharing scenario was a functional test case. However, switching sharers in conjunction with other actions, such as taking/granting control of an application-sharing session, created a condition that exposed a bug.

3.1.2 Environmental Factors

We also found that having AutoBugbash run on various people's machines gave us a large diversity of environments. Hardware specs varied greatly -- We had single, dual, and quad core CPUs, with various bits of memory. Operating Systems ranged from Windows XP through Win7, Windows 2k8 servers, and mobile/web runtime environments. Office versions would be different. All of these factors also exposed timing & interaction issues.

3.1.3 Time & Samples

Finally, running the AutoBugbash over time exposed new issues whether we used new daily builds or the same build day over day. This was the nature of timing issues – The longer the scenarios were executed, the more likely a timing issue was exposed.

3. Conclusion

The AutoBugbash project has been a great addition to our test portfolio. During the development of the AutoBugbash, we learned several key lessons:

1. Controlled Randomization – We achieved the best results through a controlled (weighted) randomization algorithm as opposed to using structured functional or completely random models.
2. Diversification of Environment – By running on machines used by engineers, we got a diversity of environmental factors such as installed & running programs, browser versions, desktop configurations, etc. that were not attainable in a practical way in a lab environment.
3. Frequent Test Execution – In a controlled randomized environment, frequent test execution yielded additional bugs as tests were running slightly different actions every time they were executed.
4. Multi-platform support made a difference – Having a post-processing verification system allowed for an easy point of convergence in a multi-platform environment. We did not have to invest in a test framework that would span across multiple platforms (mobile, desktop, PC vs. Mac, etc.)
5. Opt-in Model – As AutoBugbash is an opt-in model, we needed to quickly eliminate any and all barriers to adoption. For example, we found that Setup needs to be "one-click". We could not make any modification to the user's environment. As a "low-touch" solution, we had to provide auto-update functionality. Finally, we used a leaderboard to give recognition to those who participated.

How can others apply this model to their problem space?

There are several elements of our solution that can be applied to other problems spaces. In particular, this solution works well in the following situations:

1. Great for Multi-machine or distributed scenarios – no need to have a centralized controller or coordinator across multiple endpoints
2. Scalable model for multiplatform scenarios -- Decentralized model allows for incompatible platforms to work together

3. Great for Finite State Machine test environments where there are multiple branches in a particular state – rather than exhaust every possible branch, a weighted randomization allows you to hit a representative sample across several branches

What are the next steps for AutoBugbash's evolution?

Given our preliminary findings, we want to scale up the runs on several fronts. In terms of frequency, we would like to have tighter integration with our daily engineering processes. We aim to have an AutoBugbash run automatically on every build of the product. This scales up the frequency of the run. We want to scale up the test coverage by adding more daily users into the AutoBugbash. Finally, we want to scale up the tested functionality by continuing to add Action/Event pairs to ensure that Actions are implemented for all core functionality in the product.