Monkey Testing Revisited: Using Automated Stress Testing

Praveen Hegde, Perry Hunter, Feng Liang, Doug Reynolds
Tektronix, Inc.
P.O. Box 500, M/S 39-732
Beaverton, Oregon 97077
praveen.hegde@tek.com; perry.w.hunter@tek.com;
feng.liang@tek.com; douglas.f.reynolds@tek.com

Abstract

Have you ever thought about paying people off the street to come in and press buttons and turn knobs as a way of stress testing your user interfaces? We have joked about bringing our kids in and letting them do this. As impractical as this may be, we know that we need to test our code in a variety of ways to ensure that we have done our best to find defects before our customers do. Though this practice of manual stress testing is still being used, it has significant weaknesses. The primary weakness is once a problem is found; it is difficult or impossible to reproduce the defect because the tester was not following a pre-defined sequence of events.

This paper describes what stress testing means, what some of the benefits and weaknesses are of automating stress testing, how we implemented automated stress testing to augment other testing strategies, and what some of the lessons learned are with our experience of using automated stress testing. Our Graphical User Interface stress testing has been instrumental not only in finding problems within our software applications but also in the third party tool used to perform the stress testing.

Introduction to Stress Testing

In our organization, software testing parallels the entire software development cycle. This testing is accomplished through reviews (product requirements, software functional requirements, software designs, code, test plans, etc.), unit testing, system testing (also known as functional testing), expert user testing (like beta testing but in-house), smoke tests, etc. All these 'testing' activities are important and each plays an essential role in the overall effort but, none of these specifically look for problems like memory and resource management. Further, these testing activities do little to quantify the robustness of the application or determine what may happen under abnormal circumstances. We try to fill this gap in testing by using stress testing.

Stress testing can imply many different types of testing depending upon the audience. Even in literature on software testing, stress testing is often confused with load testing and/or volume testing. For our purposes, we define stress testing as **performing**

random operational sequences at larger than normal volumes, at faster than normal speeds and for longer than normal periods of time as a method to accelerate the rate of finding defects and verify the robustness of our product.

Stress testing in its simplest form is any test that repeats a set of actions over and over with the purpose of "breaking the product". The system is put through its paces to find where it may fail. As a first step, you can take a common set of actions for your system and keep repeating them in an attempt to break the system. Adding some randomization to these steps will help find more defects. How long can your application stay functioning doing this operation repeatedly? To help you reproduce your failures one of the most important things to remember to do is to log everything as you proceed. You need to know what exactly was happening when the system failed. Did the system lock up with 100 attempts or 100,000 attempts?[1]

Note that there are many other types of testing which have not mentioned above, for example, risk based testing, random testing, security testing, etc. A good review of different testing types is provided by Cem Kaner and James Bach[2]. We have found, and it seems they agree, that it is best to review what needs to be tested, pick multiple testing types that will provide the best coverage for the product to be tested, and then master these testing types, rather than trying to implement every testing type.

Some of the defects that we have been able to catch with stress testing that have not been found in any other way are memory leaks, deadlocks, software asserts, and configuration conflicts. For more details about these types of defects or how we were able to detect them, refer to the section 'Typical Defects Found by Stress Testing'.

Table 1 provides a summary of some of the strengths and weaknesses that we have found with stress testing.

Table 1
Stress Testing Strengths and Weaknesses

Strengths	Weakness
Find defects that no other type of test would find	Not real world situation
Using randomization increase coverage	Defects are not always reproducible
Test the robustness of the application	One sequence of operations may catch a problem right away, but use another sequence may never find the problem
Helpful at finding memory leaks, deadlocks, software asserts, and configuration conflicts	Does not test correctness of system response to user input

Background to Automated Stress Testing

Stress testing can be done manually - which is often referred to as “monkey” testing. In this kind of stress testing, the tester would use the application “aimlessly” like a monkey - poking buttons, turning knobs, “banging” on the keyboard etc., in order to find defects. One of the problems with “monkey” testing is reproducibility. In this kind of testing, where the tester uses no guide or script and no log is recorded, it’s often impossible to repeat the steps executed before a problem occurred. Attempts have been made to use keyboard spyware, video recorders and the like to capture user interactions with varying (often poor) levels of success.

Our applications are required to operate for long periods of time with no significant loss of performance or reliability. We have found that stress testing of a software application helps in accessing and increasing the robustness of our applications and it has become a required activity before every software release. Performing stress manually is not feasible and repeating the test for every software release is almost impossible, so this is a clear example of an area that benefits from automation, you get a return on your investment quickly, and it will provide you with more than just a mirror of your manual test suite.

Previously, we had attempted to stress test our applications using manual techniques and have found that they were lacking in several respects. Some of the weaknesses of manual stress testing we found were:

1. Manual techniques cannot provide the kind of intense simulation of maximum user interaction over time. Humans can not keep the rate of interaction up high enough and long enough.
2. Manual testing does not provide the breadth of test coverage of the product features/commands that is needed. People tend to do the same things in the same way over and over so some configuration transitions do not get tested.
3. Manual testing generally does not allow for repeatability of command sequences, so reproducing failures is nearly impossible.
4. Manual testing does not perform automatic recording of discrete values with each command sequence for tracking memory utilization over time – critical for detecting memory leaks.

With automated stress testing, the stress test is performed under computer control. The stress test tool is implemented to determine the applications’ configuration, to execute all valid command sequences in a random order, and to perform data logging. Since the stress test is automated, it becomes easy to execute multiple stress tests simultaneously across more than one product at the same time.

Depending on how the stress inputs are configured stress can do both ‘positive’ and ‘negative’ testing. Positive testing is when only valid parameters are provided to the device under test, whereas negative testing provides both valid and invalid parameters to the device as a way of trying to break the system under abnormal circumstances. For example, if a valid input is in seconds, positive testing would test 0 to 59 and negative testing would try –1 to 60, etc.

Even though there are clearly advantages to automated stress testing, it still has its disadvantages. For example, we have found that each time the product application changes we most likely need to change the stress tool (or more commonly commands need to be added to/or deleted from the input command set). Also, if the input command set changes, then the output command sequence also changes given pseudo-randomization.

Table 2 provides a summary of some of these advantages and disadvantages that we have found with automated stress testing.

Table 2
Automated Stress Testing Advantages and Disadvantages

Advantages	Disadvantages
Automated stress testing is performed under computer control	Requires capital equipment and development of a stress test tool
Capability to test all product application command sequences	Requires maintaince of the tool as the product application changes
Multiple product applications can be supported by one stress tool	Reproducible stress runs must use the same input command set
Uses randomization to increase coverage; tests vary with new seed values	Defects are not always reproducible even with the same seed value
Repeatability of commands and parameters help reproduce problems or verify that existing problems have been resolved	Requires test application information to be kept and maintained
Informative log files facilitate investigation of problem	May take a long time to execute

In summary, automated stress testing overcomes the major disadvantages of manual stress testing and finds defects that no other testing types can find. Automated stress testing exercises various features of the system, at a rate exceeding that at which actual end-users can be expected to do, and for durations of time that exceed typical use. The automated stress test randomizes the order in which the product features are accessed. In this way, non-typical sequences of user interaction are tested with the system in an attempt to find latent defects not detectable with other techniques.

To take advantage of automated stress testing, our challenge then was to create an automated stress test tool that would:

1. Simulate user interaction for long periods of time (since it is computer controlled we can exercise the product more than a user can).

2. Provide as much randomization of command sequences to the product as possible to improve test coverage over the entire set of possible features/commands.
3. Continuously log the sequence of events so that issues can be reliably reproduced after a system failure.
4. Record the memory in use over time to allow memory management analysis.
5. Stress the resource and memory management features of the system.

Typical Defects Found by Stress Testing

At Tektronix we have found that stress testing has been successful at finding asserts, memory leaks, deadlocks, and resource problems, etc. We have found different techniques for identifying and reproducing these defects. The following section defines these types of defects.

A program application assertion occurs when something abnormal in the application occurs that the software designer believes should not happen. Typically when a program assertion occurs the program halts and no further operations may be executed. An assertion is a statement that is considered to always be true. It is this statement that is used as a check against the code to demonstrate program consistency. An assertion statement will not be executed under normal circumstances.

A program application deadlock occurs when two processes are holding resources that each other require. A deadlock situation is not likely to occur under normal circumstances. However, putting the software application under stress is likely to cause a deadlock to occur. A deadlock usually manifest itself by no longer executing commands and/or overflowing input/output buffers as new command requests continue to be exercised.

A program application memory leak is the gradual loss of available memory when the program application repeatedly fails to return memory that it has obtained for temporary use. As a result, the available memory for that application becomes exhausted and the program application begins to slow down or no longer functions. For a program that is frequently opened or called or that runs continuously, even a very small memory leak can eventually cause the program to begin to slow down or terminate. A memory leak is the result of a program defect.

Other resource problems can also occur, for example what happens when the hard disk becomes full? How does the system react under this condition? We have found that there are many other resource problems that can occur within a system. Many of these resource problems can cause adverse effects like page faults, core dumps, etc. which cause the program to terminate unexpectedly.

~~6.Totally distort the normal order of processing, especially processing that occurs at different priority levels.~~

7. ~~Force the exercise of all system limits, thresholds, or other controls designed to deal with overload situations.~~
8. ~~Greatly increases the number of simultaneous actions~~

Requirements for an Automated Stress Test Tool

There are many requirements that could be added to an automated stress test tool but the following is a list of the essential and desired requirements for an automated stress tool. Though some of these requirements have been mentioned before, we will re-hash them here for completeness and further describe some of the necessary attributes for each of the requirements.

Essential requirements of an Automated Stress Test Tool include:

- **Reproducible Command Sequences:** The stress test tool should be able to produce a random yet repeatable series of command instructions. This could be done using a pseudo-random generator, which uses a unique seed value to create a unique sequence of command instructions to be executed.
- **Command Sequence Logging:** Each action executed by the system should be logged in a sequential file, ideally, the file can be played back in whole or in part to reproduce any problems that occurred during the test. The file should also be able to track the seed number and the number of commands executed.
- **Memory and Resource Utilization Monitoring:** At discrete intervals during the automated stress testing, values representing the free and allocated memory and other resources available in the system should be recorded. This allows for memory leaks and resource depletion issues to be identified.
- **Fault Tolerance:** The automated stress test should be able to handle minor failures when exercising the product application. The automated stress test tool should be able to gracefully handle the fault situation, log messages as needed and continue with the test without significant reduction in the rate at which commands are sent to the system, and without itself impacting the available memory, resources or ability of the DUT to respond to simulated user inputs. On the other hand, if a deadlock condition occurs, then the stress test should stop.

Desirable requirements of an Automated Stress Test Tool include:

- The capability to pause the stress test after a set number of command sequences. This allows the checking of status or provides the ability to run the stress test to just before the point where the DUT fails where the remaining operation(s) may be performed in a single step mode or entered by hand while the system is being debugged.
- The capability to set a delay time between command sequences to support different application speeds, i.e. accessing pop-up menus on one application may take longer than another application.

- The capability to increment the pseudo random seed number every set number of command sequences and store the state of the DUT, for example the stress test could run for 500 command sequences, save the state of the DUT, increment the seed number and then run for another 500 commands sequences.
- The capability to execute the command set sequentially (this is mainly used for debugging purposes which can be used to check that the command set is entered correctly and are valid).
- The capability to generate feedback while the test is running so the state of the test can be checked at anytime (test application identifier, number of commands executed, runtime, current seed value, etc).
- The capability to randomize input values within a command. For example, if a command accepts a parameter from 1 to 4 then any of the numbers 1, 2, 3, or 4 should be selectable in a pseudo-random fashion.
- The capability to recognize the test application configuration and make adjustments accordingly.

Given these requirements for an automated stress test tool, our goal was to develop tools that could be used to augment our existing test methods, fulfill the stated requirements and fill the gaps to allow us to provide products that meet or exceed our customers' expectations.

Automated Stress Testing Implementation

Automated stress testing implementations will be different depending on the interface to the product application. At Tektronix, the types of interfaces available to the product drive the design of the automated stress test tool. The interfaces fall into two main categories:

1) **Programmable Interfaces:** Interfaces like command prompts, RS-232, Ethernet, General Purpose Interface Bus (GPIB), Universal Serial Bus (USB), etc. that accept strings representing command functions without regard to context or the current state of the device.
2) **Graphical User Interfaces (GUI's):** Interfaces that use the Windows model to allow the user direct control over the device, individual windows and controls may or may not be visible and/or active depending on the state of the device.

At Tektronix, many of our products also have front panels with actual buttons and knobs. Many of these products also have 'backdoor' programmable access and/or alternative interfaces (like USB), which simulate the actual button pushes and knob turnings. Granted this is not quite the same as actually pushing the mechanical button or twisting knobs but we are able so simulate these and test the software's reactions.

Programmable Interfaces

Tektronix has been using programmable interfaces for many years (starting in the 1970's). These interfaces have allowed users to setup, control, and retrieve data in a variety of application areas like manufacturing, research and development, and service. To meet the needs of these customers, the products provide programmable interfaces, which generally support a large number of commands (1000+), and are required to operate for long periods of time, for example, on a manufacturing line where the product is used 24 hours a day, 7 days a week. Testing all possible combinations of commands on these products is practically impossible using manual testing methods.

Programmable interface stress testing is performed by randomly selecting from a list of individual commands and then sending these commands to the device under test (DUT) through the interface. If a command has parameters, then the parameters are also enumerated by randomly generating a unique command parameter. By using a pseudo-random number generator, each unique seed value will create the same sequence of commands with the same parameters each time the stress test is executed. Each command is also written to a log file which can be then used later to reproduce any defects that were uncovered.

For additional complexity, other variations of the automated stress test can be performed. For example, the stress test can vary the rate at which commands are sent to the interface, the stress test can send the commands across multiple interfaces simultaneously, (if the product supports it), or the stress test can send multiple commands at the same time.

Graphical User Interfaces

In recent years, Graphical User Interfaces have become dominant and it became clear that we needed a means to test these user interfaces analogous to that which is used for programmable interfaces. However, since accessing the GUI is not as simple as sending streams of command line input to the product application, a new approach was needed. It is necessary to store not only the object recognition method for the control, but also information about its parent window and other information like its expected state, certain property values, etc. An example would be a 'HELP' menu item. There may be multiple windows open with a 'HELP' menu item, so it is not sufficient to simply store "click the 'HELP' menu item", but you have to store "click the 'HELP' menu item for the particular window". With this information it is possible to uniquely define all the possible product application operations (i.e. each control can be uniquely identified).

Additionally, the flow of each operation can be important. Many controls are not visible until several levels of modal windows have been opened and/or closed, for example, a typical confirm file overwrite dialog box for a 'File->Save As…' filename operation is not available until the following sequence has been executed:

1. Set Context to the Main Window
2. Select 'File->Save As…'
3. Select Target Directory from tree control

4. Type a valid filename into the edit-box
5. Click the 'SAVE' button
6. If the filename already exists, either confirm the file overwrite by clicking the 'OK' button in the confirmation dialog or click the cancel button.

In this case, you need to group these six operations together as one "big" operation in order to correctly exercise this particular 'OK' button.

Data Flow Diagram

A stress test tool can have many different interactions and be implemented in many different ways. Figure 1 shows a block diagram, which can be used to illustrate some of the stress test tool interactions. The main interactions for the stress test tool include an input file and Device Under Test (DUT). The input file is used here to provide the stress test tool with a list of all the commands and interactions needed to test the DUT.

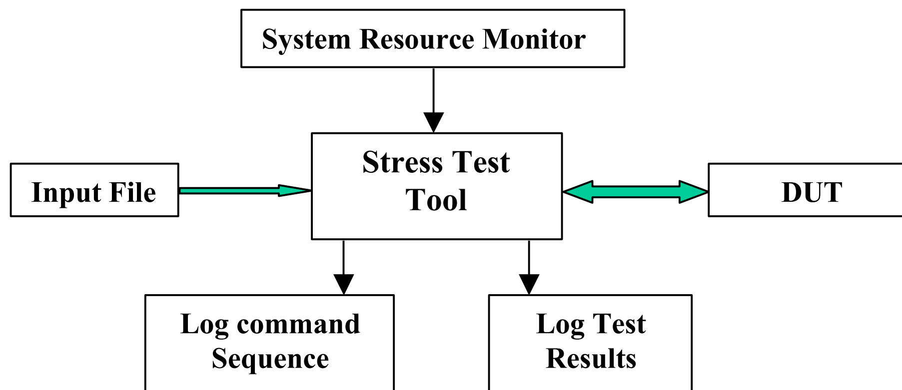

Figure 1: Stress Test Tool Interactions

Additionally, data logging (commands and test results) and system resource monitoring are very beneficial in helping determine what the DUT was trying to do before it crashed and how well it was able to manage its system resources.

The basic flow control of an automated stress test tool is to setup the DUT into a known state and then to loop continuously selecting a new random interaction, trying to execute the interaction, and logging the results. This loop continues until a set number of interactions have occurred or the DUT crashes. The following c-code is an example of the basic structure of an automated stress test tool.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

        long    seed    = 1;                            // Initialize random number seed

short random_number (void) {
        int     increment= 1;                   // Random number increment value
        int     multiplier = 0x15a4e35L;        // Magic number for random number generator
```

```
        seed = multiplier * seed + increment;      // Generate new seed value (i.e. random #)
        return ((short) (seed >> 16) & 0x7fff);
} // END random number generator



int main (int argc, char **argv) {
        char        buffer [5][32];

        int         command;                              // Interaction to select from
        int         commands;                             // Total interactions to choose from
        int         count              =  0;              // Number of commands executed
        int         total              = 10;              // Number of commands to execute


        // create dummy array of interactions
        commands = 5;
        sprintf (buffer[0],"%s", "do file close");
        sprintf (buffer[1],"%s", "do file delete");
        sprintf (buffer[2],"%s", "do file open");
        sprintf (buffer[3],"%s", "do file print");
        sprintf (buffer[4],"%s", "do file save");

        // initialize random number generator and DUT to a known state
        do {        // Loop until total interactions have occurred or instrument crashes
                    // Generate which command to select
                    command = (int)((((double)(random_number() * commands)) / 32767.0) + 1.0);

                    count++;
                    // Send command to interface and log to data file; check application crash
                    // Here we will just use a printf statement
                    printf ("Command %2d is:  %s\n", count, buffer [command -1]);

                    _sleep (0);                           // Delay time between sending commands

                    // If system resources are to be logged that could be done here
        } while (count < total);                  // Only send total commands
}                                                         // END of Example Program
```

This c-code example will output “do file …” in a pseudo-random sequence. If the seed value and command buffer are not changed then the program output sequence will be the same every time the program is run. If the seed value were to be changed, then the program output sequence will also change. Likewise, if commands are added to or deleted from the buffer, even the same seed will generate different sequences of commands.

Programmable Interface Stress Test Tool

Tektronix has been using a programmable interface stress test tool for many years. This tool has proven to be highly valuable in catching obscure faults in the DUT that would be difficult or impossible to find using other test methods. Things like assertion failures, memory leaks, pointer problems and the like have been detected and fixed where previous to the implementation of the tool, and these defects would likely have passed through the testing process. Indeed, the tool has uncovered defects that have existed in the field for some time, but are either obscure or so difficult to reproduce that users have never reported them.

The programmable interface stress test tool uses log files, which record the sequence of commands, applied to the product and have been necessary to analyze system failures since the failure may have actually occurred earlier (for example, a memory

leak may be detected after a particular sequence of operations, or over a long period of time). One item that was not in our early versions of stress was the ability to track system resources like memory. This has since become a requirement.

The stress test tool command sequences should also be able to handle randomization of parameters within a command, for example if a command takes a parameter of type integer then the stress test should be able to select an integer value within a pre-defined range and reproduce the value on a subsequent re-test. There are many types of input parameters some numeric (like integers and real numbers), some are alphanumeric (like names, labels, etc), and some are enumerations (like selections). We also have also found that supporting macros and other complex algorithms has helped us instantiate random complex math expressions and the like.

The stress test tool should also have some idea on how to terminate on a failure. For the most part a timeout value can be used. The timeout value is the value in which the program application must respond before it is considered to have had a failure. Since each program application is different the timeout value is generally selectable at runtime.

Graphical User Interface Stress Test Tool

Though many of the requirements for GUI stress testing were identical to those for programmable interface stress testing, we found ourselves making design decisions based upon the capabilities of the third party test automation tool. Where our programmable interface stress test tool uses a flat-file listing of programmable interface commands, the GUI stress test tool uses various object recognition methods to identify and act upon each control in the GUI. Each control had its own context defined by the window that contained it, and therefore it was possible to have objects share non-unique object recognition methods. We decided that the best way to store and use this information was to use a database. With the information stored in the database, not only could a unique command sequence be generated but also unique parameters for the enumeration of buttons, checkboxes, or numeric/alphanumeric input fields.

We augment the third party test tool by creating our own library functions. For example, while the test is running, we use custom library functions to handle the output of the command sequences to one log file and the test results to another log file. At the same time, values for critical memory and resource values are captured using API functions and written to the test result log file. One advantage we have found to using our own logging functions is that we have control of these log files when the system crashes and do not lose data due to log database corruption in the third party test tool.

Lessons Learned with GUI Automated Stress Testing

This section points out some of the lessons we have learned from GUI based automated stress testing including tips for design and development, early stages of stress testing, problems found through test execution, and techniques used to isolate defects.

Design and Development

During the development of our GUI-based stress test tool, several issues arose that had not been fully considered at the outset. These were:

Object Recognition

The third party test automation tool could not recognize the custom tightly coupled (embedded) OCX controls in our product applications. This necessitated the development of a custom object recognition library that required significant effort to implement. We also learned that early and well thought-out effort to create an object-recognition map would pay large dividends in reduced maintenance throughout the project. A table (Object Recognition Map) could be used to assign unique recognition information for each object in the product application. Changes made to a single definition of an object can then propagate throughout the code with little or no effort. A crud example of Object Recognition Map using Notepad would be to add a layer of abstraction for the 'File->Save' operation. To do this the 'File->Save' operation would be like:

```
#define Notepad_File_Save "File->Save"
```

By de-referencing the 'File->Save' operation to Notepad_File_Save, if the 'File->Save' operation were to change then only the object recognition map would have to be changed. All the underlying test case code would remain the same since it is using Notepad_File_Save. This is very handy while the underlying GUI interface is still under development but the underlying product features are concrete.

Stress Input Data Source Control

The ability to go back and reproduce the same set of operations, using the same input file and seed value, must be maintained. Because of this, we have found that we must keep the input file/database in a configuration management system.

Inadequate Test Logs

The test log capability provided by the third party tool generated as output during script execution was inadequate for the examination of memory and resource values, executed command sequences and other data that would have been beneficial in analyzing the applications behavior during and after the test. Additionally, in the event of a test-tool related failure or system crash, the log file was left in an incomplete state, and at times was rendered inaccessible due to file corruption. Therefore, a custom set of logging functions was developed that collected the necessary data and wrote it off to external files for later analysis.

Early Stages of Stress Testing

Typically when we first start stress testing on a new DUT in the early stages of integration, the stress test may only run for a short period of time. To help isolate faults as each new subsystem is integrated into the application we, will run stress testing for that subsystem only. Once the subsystem becomes stable, we add the commands for that subsystem to the main stress data input file and run the integrated set of

commands. This process continues until all subsystems have been individually added to the main stress data input file, allowing the integrated DUT to be fully tested.

The following graph in Figure 2 shows how the number of commands in a given stress test of the DUT increased over time. The flat region of the chart (< 1000 commands) roughly represents the period when integration build was adding new subsystems to the DUT, while the region in which the number of commands increases geometrically represents the period when the system was fully integrated and more difficult-to-detect issues were found.

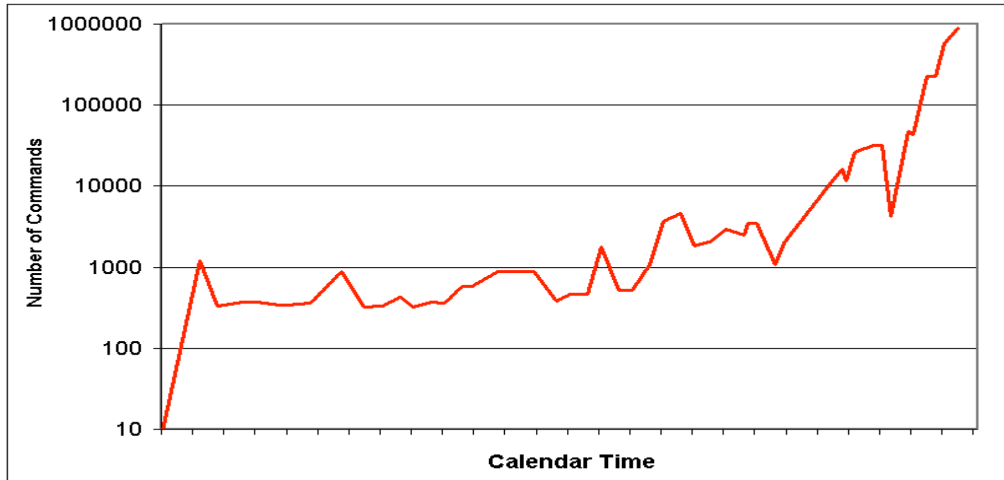

Figure 2: Application Stress Testing Over Time

Execution

Third Party Tool Memory Management Issues

During execution of GUI stress we found that the third party tool had serious memory management issues that would occur when the tests were run. We found that each time we would run our tests, the third party test tool would leave blocks of memory open causing increased swap file usage, which in time increased the time it took to execute a command sequence. Left unchecked, the system would hang or crash well before useful data from the DUT could be obtained. Also, the memory loss from within the third party tool tended to mask memory and resource management issues from within the DUT.

We isolated the memory problems within the third party tool by running them against known 'good' applications like Notepad®. We did this by creating an data input file for the application and then use the stress test tool to stress the application while monitoring the system resources. We found that each time certain stress test tool error conditions were encountered; it left blocks of memory open. We resolved the memory losses within the stress test tool by working with the vendor to fix the defects over time,

however, great care needs to be taken to ensure that the stress test tool can work gracefully under the test conditions.

Test Execution Time was Variable

As the stress test runs for longer durations of time, it was noticed that the overall performance decayed (related to memory management issues mentioned above), which lead to increasing time for display of GUI objects. Given sufficient run length, the rate at which commands could be delivered to the system dropped well below what was possible for a normal user to generate. The situation was improved, but not entirely eliminated by implementing a "wait for object" structure in place of the original "wait for time delay" to prevent the script from overrunning the DUT application.

Techniques Used to Isolate Defects

Depending on the type of defect to be isolated, two different techniques are used:

1. System crashes – (asserts and the like) do not try to run the full stress test from the beginning, unless it only takes a few minutes to produce the defect. Instead, back-up and run the stress test from the last seed (for us this is normally just the last 500 commands). If the defect still occurs, then continue to reduce the number of commands in the playback until the defect is isolated.
2. Diminishing resource issues – (memory leaks and the like) are usually limited to a single subsystem. To isolate the subsystem, start removing subsystems from the database and re-run the stress test while monitoring the system resources. Continue this process until the subsystem causing the reduction in resources is identified. This technique is most effective after full integration of multiple subsystems (or, modules) has been achieved.

Some defects are just hard to reproduce – even with the same sequence of commands. These defects should still be logged into the defect tracking system. As the defect re-occurs, continue to add additional data to the defect description. Eventually, over time, you will be able to detect a pattern, isolate the root cause and resolve the defect.

Some defects just seem to be un-reproducible, especially those that reside around page faults, but overall, we know that the robustness of our applications increases proportionally with the amount of time that the stress test will run uninterrupted.

Future Improvements

We are continuously looking at ways to improve our stress test tools. Our GUI stress tool still relies on a third party application as the front end to perform object recognition. We are working on moving away from this dependency by developing an in-house stand-alone stress test tool. This will increase the number of platforms on which the stress test can be run since we will no longer be limited to those supported by the third party application.

Another area for improvement includes standardizing the data input file. Currently our programmable interface stress test tool uses a standard database file that works with all of our applications. We are now applying the same constraints to the GUI stress test tool. This will make it easier to support more applications with fewer stress test tool changes.

Last, we are considering a control-sequencing application that would recursively inspect the DUT to create our database. Currently, it takes significant effort to create a new database since all the control attributes must be entered by hand.

Summary

Our company has a heritage of delivering products of highest quality and reliability. One of the methods we use to ensure this high quality and reliability is through stress testing. Manual stress testing has proven to be very ineffective or impractical. We created an automated GUI stress test tool using a model similar to our existing programmable interface stress test tool that has been effective for many years. Automated GUI stress testing has proven to be an effective way to find elusive and difficult to reproduce defects that were not identified by any other means.

References

[1] Mark Fewster & Dorothy Graham, “Software Test Automation”, Great Britain, Addison-Wesley 1999, pp548-550.

[2] Cem Kaner & James Bach, Software Testing, Analysis & Review Conference (STAR West), October, 1999, Reviews different approaches to the design of software tests.

Glossary

Command Set	A set of all commands to be exercised by the stress test tool.
Command Sequence	A sequence of commands generated from a command set using a pseudo-random generator.
DUT	Device Under Test. The application software/hardware to be tested.
GPIB	General Purpose Interface Bus, also referred to as IEEE Standard 488.
GUI	Graphical User Interface.
Load Testing	A type of testing which evaluates how will a system operates under extreme load conditions. Used to test the system capacity.
PI	Programmer Interface is a set of commands that are used to develop custom applications that interact with the DUT.

RS-232	Interface between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange, also referred to as EIA Standard RS-232-C.
Stress Testing	A type of testing which is used to perform random operational sequences at larger than normal volumes, at faster than normal speeds and for longer than normal periods of time as a method to accelerate the rate of finding defects and verify the robustness of our product.
Volume Testing	A type of testing that is used to expose a system to large volumes of data.

About the Authors

Praveen Hegde is a graduate of India's Govt. BDT College; he holds a degree in Electronics & Communication Engineering. He has worked for the past 5 years within the areas of Safety & Mission Critical Software Testing, GUI Testing, and Embedded Systems Testing.

Perry Hunter is a Software Quality Engineer with Tektronix Inc., living in Portland Oregon. He is a graduate (1987) of California's Humboldt State University and has worked in software development and test in a variety of roles for approximately 15 years.

Feng Liang graduated from ASU (Arizona State University, Tempe) in 1998 with his Master of Computer Science degree. He worked at Tektronix Inc. as Software Quality Engineer for the past 3 years. His responsibilities include automated testing and software process improvement.

Doug Reynolds is a Software Quality Engineer at Tektronix with the Instrument Business Unit. He has worked for Tektronix for the past 10 years. He is currently working on a Master of Computer Science and Engineering degree at Oregon Health & Science University (OHSU).