Experiments with High Volume Test Automation

Pat McGee
Florida Institute of Technology
Computer Sciences Department
150 W. University Blvd.
Melbourne, FL 32901
321-409-5521
jpmcgee@cs.fit.edu

Cem Kaner
Florida Institute of Technology
Computer Sciences Department
150 W. University Blvd.
Melbourne, FL 32901
321-674-7137
kaner@kaner.com

ABSTRACT

We are working with a broad class of testing techniques we collectively call High Volume Test Automation (HVTA). The essence of HVTA techniques is automated execution and evaluation of large numbers of tests, for the purpose of exposing functional errors that are otherwise hard to find. These techniques are not widely used in industry, but we believe they have the potential to help us substantially increase the reliability of software.

We propose to find existing industry HVTA projects, write informal case studies of them, create our own tools to implement the technique, and apply our tool in a case study of one or more open source projects.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools..*

General Terms

Reliability, Security, Verification.

Keywords

Software Testing, High Volume Test Automation, Regression Testing.

1. INTRODUCTION

We are working with a broad class of testing techniques we collectively call High Volume Test Automation (HVTA). The essence of HVTA techniques is automated execution and evaluation of large numbers of tests, for the purpose of exposing functional errors that are otherwise hard to find.

We believe HVTA techniques can find certain types of errors much better than most traditional test techniques. These types include buffer overruns, special cases, timing related errors, corrupt memory or stack, memory or resource leaks, and resource exhaustion errors. In particular, we believe that, by using HVTA techniques, people can dramatically increase the reliability of software that must run for long periods of time without stopping or rebooting.

While HVTA techniques are better at finding these types of errors, they are usually not better at finding several other types of errors. Requirements errors are often better found by other techniques such as formal methods, user testing, or exploratory testing. Simpler functional errors are often better found by domain testing, boundary value analysis, etc. HVTA techniques can find these errors, but the other techniques are usually less expensive to apply.

In particular, HVTA techniques are much more effective on code that is already fairly stable and passes basic functional tests. For such code, we believe it can lead to substantial increases in reliability.

High volume automated testing has been used to qualify safety-critical software, such as air traffic control systems, medical heart monitors, and telephone systems. It has also been used to qualify firmware in consumer and office automation products, such as printers and networking equipment, and to expose faults in software such as word processors and operating systems.

There is very little empirical research on HVTA techniques. Much of the development that has been done has been proprietary industrial development. We propose to do a reverse technology transfer, from isolated pockets in industry to the academy and the public.

2. RESEARCH ACTIVITIES

We plan to perform our research in several stages. We have started on the first two of these.

2.1 Capture an industry experience

The first stage is to perform an informal descriptive case study [7] of a specific HVTA technique in an industrial setting. We will capture enough information to understand the technique, how it was used, the overall pattern of results, the technique users' beliefs about the types of errors it is effective at exposing, and some of its limitations. In many cases, we will work behind a non-disclosure agreement that allows us to describe the technique, the internal tools that implement the technique, and to hint at results. We would likely not be able to detail specific results or sometimes even disclose the name of the company or product. These will contain a level of detail usually found in experience reports rather than formal case studies.

We have completed one of these reports. [4] So far, we have not found a forum in which to publish this work. Reviewers correctly point out that non-disclosure agreements prohibit including many of the details that would make this work of interest to academic audiences.

2.2 Create a tool

We will use these less formal reports to inspire further development and more formal case studies. We plan to develop our own tools to implement the technique. We plan to release these tools as open source products. We have started on one such tool to implement Extended Random Regression testing [4]

Given that the tool will be freely available, we expect that other people will apply the tool to their own specific projects. We hope that this will encourage some of them to perform their own case studies or to allow us or other academic researchers to use them as subjects of formal case studies.

2.3 Apply the tool in a descriptive case study

We then plan to use that tool to perform a more formal case study. In this case study, we will apply the tool to one or more other open source projects. In this case study, we will serve both as the Experimenter and as the Subject.

It can be hard to find good subject programs to test. We propose to use large programs developed by the open source community. These programs are freely available and have publicly available bug tracking systems. Examples of programs we plan to test are Mozilla and Open Office.

We will use these projects developed by other people, but we will apply the tool ourselves, rather than collect data as other subjects apply the tool to their own projects. Because we will operate on open source products, the source code, the tests, and the bug database will all be publicly available, allowing others to much more easily validate our data.

While creating our tools and running them against these programs, we will collect time and effort data, both while creating the tool and while using it to test the product. That data, combined with the bugs submitted to the developers and their evaluations and responses to the bugs, should allow us to derive efficiency measures.

3. EXPECTED OUTCOMES

3.1 Outcome variables

We expect to primarily measure effort in applying the test technique, number, type, and severity of errors found, and improvements in reliability of the end product as a result of applying the technique. We plan to report bugs that we discover back to the developers, and to include their feedback and responses to these bugs in our evaluation of the test technique and the tool.

Each development team that might use HVTA techniques will of course have their own specific outcome variables that they treat as important. User satisfaction is important to product development teams. For some, an interesting variable will be efficiency in the test process. For others, it may be the MTBF of the product. For others, it might be reduction in technical support costs. For still others, it might be the reduction in risk and in contingent liabilities due to possibility of lawsuits.

While we recognize that these are important, we also recognize that they are very dependent on the specific context of the user. We will not be able to predict or measure most of these. So, we will primarily concentrate on those outcome variables that are common to many contexts, even though they may be less important than others to specific potential users.

3.2 Threats to external validity

We hope to achieve results that generalize well by performing tests on real programs currently under development.

One threat to external validity of these case studies is that the products we will test will all be open source development efforts. These projects tend to be somewhat different from most proprietary development efforts. They tend to have a larger number of developers. The developers are often not driven by the goal of being profitable, and so make different engineering tradeoffs. There may or may not be a bias causes by different individual viewpoints or skills. The projects tend to use methodologies with shorter development cycles.

Therefore, the results will be less generalizable to some other types of development, specifically projects developed with waterfall model and long development cycles, and with the goal of being economically profitable in both the short and long terms.

Another possible threat might be the dependency on specific type of product. We plan to address this threat by testing several different types of products, including operating systems, desktop applications, back-end applications, and embedded software.

We will be performing both as the Experimenter and as the Subject, while and after writing the tool. This will tend to make the case study less generalizable. We will of course become expert in the tool, since the same team will be writing the tool and using it to test another product. We hope to address this threat by recruiting external testers that will allow us to study their use of the tool.

4. PROBLEMS POTENTIALLY FOUND

Here are some of the types of problems and some HVTA techniques that can be used to expose them.

4.1 Buffer overruns and security holes

Start with a properly formatted data file of a type commonly available on the web, such as Word, PDF, Flash, or Real Audio. Use HVTA (in this case, “hostile data stream testing” [2]) to corrupt the file, trying thousands or millions of variations of the original file. The goal is to create a file that will (a) look acceptable to the program that reads it, and (b) overflow some internal buffer. Then, (c) exploit this failure to execute a follow-up program on the target computer

4.2 Special cases

Some failures occur only on the occurrence of a narrow condition. Some narrow conditions (e.g., divide by zero) are obvious and tested routinely. Others are much more subtle. The Intel FDIV bug occurred on only a very small percentage of possible input cases. [5]. Edelman [1] shows that if you understand the mathematics of this particular bug, there are some efficient ways to expose it. However, in the face of arbitrarily many different types of bugs, there’s something to be said for a brute force attack that simply tries a massive number of calculations, comparing the results against those from a known good set of reference functions.

4.3 Timing related errors

Race conditions are common in telephone systems under development and several were exposed in the simulator-based high-volume testing done by Telenova, a PBX manufacturer, described in [3]. They have also been implicated in the 2003 northeast US power blackout. [6]. Using Extended Random Regression testing exposed a significant number of these in "Mentsville" [4]. Running very large numbers of test cases without restarting or resetting the program can be useful in finding this type of problem.

4.4 Corrupt memory or stack

An event puts bad data into some memory location. At some later, possibly much later time, another function attempts to read and act on the bad data. Wild pointers are a classic example of this problem. For some well-known platforms, there are useful tools to expose these problems, but they are not available for all platforms (including the largely custom ones) used to develop embedded software. Running the program for a long time with large numbers of test cases can help find this type of problem.

4.5 Memory or resource leaks

Some memory leaks are extremely difficult to pin down. For example, consider a system that recovers unused memory via garbage collection. The amount of apparently-free memory decreases with normal operation of the program and is restored to maximum by garbage collection. This maximum varies depending on what functions are active and how they are using memory at a given time. A decline in free memory from one garbage collection to the next might reflect entirely normal error-free operation. A memory leak might not be apparent until a declining trend is seen after many garbage collections, and by then, it might be difficult to determine which function has the problem.

4.6 Resource exhaustion

Some sequences of operations might exhaust some resource, such as stack frames or file descriptions, while many other sequences through the same code might not. If traditional tests do not test any of the few sequences that do consume more resources than are available, they will not expose those defects. Running weaker tests on more of the possible sequences can raise the probability of exposing these defects.

5. Current status

We have performed one case study on a test technique used in a few places in industry, Extended Random Regression testing[4]. We have found no academic references to it. In this case study, we were able to gather significant history on the use of the technique. We were able to obtain permission to publish some of the details about the testing technique, the test tool, and some of the results. We were not able to obtain permission to publish details that would identify the company or the product. There are many questions that a case study should answer that we were not able to provide answers to.

The problems with not being able to publish full details of this experience led us to propose our current research plan. This plan will require somewhat more work, but will result in case studies for which we can publish all the details.

We are currently working on our own tool inspired by Mentsville's tool. We plan to make it publicly available, along with details of the experiment we will perform using the tool.

In addition, for the hostile data stream testing technique, we have written a draft-quality version of a tool and done preliminary experiments using it.[2]

6. ACKNOWLEDGMENTS

This work has been partially supported by grants from NSF (EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers."), and Rational Software / IBM, We greatly appreciate this support.

We wish to thank our first case study subject, who wishes to remain anonymous. We refer to them as Mentsville.

7. REFERENCES

[1] Edelman, A. The mathematics of the Pentium division bug. SIAM Reviews, 39 (1). 54-67.

[2] Jorgensen, A.A. Testing with Hostile Data Streams. ACM Software Engineering Notes, 28 (2).

[3] Kaner, C. The impossibility of complete testing Software QA, 1997, 28.

[4] McGee, P. and Kaner, C., Extended Random Regression Testing: Running long sequences of already-passed tests. in submitted to International Symposium on Software Testing and Analysis, (2004).

[5] Nicely, T.R. Letter to Intel: Bug in the Pentium FPU, 1994.

[6] U.S.-Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada, 2004.

[7] Yin, R.K. Case Study Research: Design and Methods, 1994.