On “Automated” Testing

Abhik Roychoudhury
National University of Singapore (NUS)
abhik@comp.nus.edu.sg

White-paper from Test1080, a spinoff from National University of Singapore

The market-place is replete with products which are performing “automated” testing of computer programs. Moreover, with the increased mobile penetration, there is a need as well as demand for automated testing of mobile apps. When a product claims to conduct “automated” testing, exactly what degree of automation exists in the market place? What does it to take to perform fully automated testing? Last but not the least, are there any conceptual difficulties or innovations, specifically needed for automated testing of mobile apps? This white-paper examines these issues. I try to make this white-paper fully self-contained for a gentle and somewhat informal introduction to fundamental issues in software testing, without having to resort to copious references.

To explain and demystify the concepts in testing, we need a few notions, which we now proceed to define rather informally. A *test-case* can be, in its simplest incarnation, a sample input to stress the behaviour of a program. For each test-case, there is a notion of an ***oracle*** which denotes the expected or intended behaviour. If the observed behaviour matches the intended behaviour, then we say that the test passes, otherwise the test fails. A *test-suite* is a set of test-cases, often with certain well-known desirable properties like statement coverage or branch coverage. Thus, if a test-suite T satisfies 95% *statement coverage*, it simply means that for 95% of the statements in the program, at least one of the test-cases in T execute the statement.

There also exist lot of terminologies both in academic and in commercial literature on bug/ error / fault / vulnerability. I do not resort to a detailed discussion on the differences between these terminologies in this introductory whitepaper. Perhaps we can grapple with them in subsequent editions. For now, let us assume that we are dealing with an “error” where the actual execution of the program differed from what was intended (also commonly called as the *intended program behaviour*) and this difference became apparent or observation through the execution of at least one test case. The aim of testing is to find such program errors.

As can be guessed, the notion of coverage simply means that more of the program elements are executed. However, mere execution of the program elements is no guarantee for fault detection. As a result, the notion of coverage simply gives us more confidence that the program behaviour has been systematically stressed but does not give us any guarantees.

Coverage is not enough

Let us proceed to establish the distinction between error execution and error detection via testing. Consider the following schematic program.

```
input in;
if (in > 0){
       if (in > 3){    //  should be  in > 4
              out = x;
       } else{
              out = 1;
       }
} else{
      out = 0;
}
output out;
```

For the purposes of the readers, I have marked the erroneous location in red, and also shown how the code differs from intended behaviour. In real-life, we are trying to find out such errors and erroneous statements through testing, and we do not know the erroneous statement in advance. Let us now give this tiny program of ours a little bit of workout, no doubt by our favourite activity, namely testing.

Suppose we execute the program with a test-case *in == 1000*. Note that the erroneous statement in > 3 is executed by this test, and yet the output of the program is 1000, as expected. In other words, we did not find or detect any errors by running the test-case *in == 1000.*

Let us now go further and execute a test-suite, a set of test-cases. Consider the test-suite {in== -1000, in == 1, in == 1000}. This test-suite even satisfies 100% statement coverage. In other words, each statement in the program is executed at least once by at least one of these three test cases. Yet, no error is detected, and all the three tests pass because they behave as "expected".

What went wrong? In this case, the "error" will be only exhibited by one single test-case in == 4. If we do not try out this test, the error will never be observed and/or detected. Even though this example is a simple one, note that having complex and nested conditionals are common in any reasonable code-base. In this case, the above code could be part of a sub-routine deep down in the code-base, so that the details of the underlying logic may not have surfaced at the level of program requirements. As a result the value in == 4 may not be tried out as a prominent use-case if we construct test-cases based on software requirements.

Automated Test generation vs. Automated Test execution

The above discussion tells us about the need to go beyond conventional measures such as statement coverage. This brings us to the focal point of this white-paper, the distinction between test generation and test execution.

As we can see from the above example program – the key difficulty is in synthesizing the "right" test-case in == 4, so that the "error" in the program gets exposed. Once the test cases are given, executing the tests is simply assumed to be automated.

Thus, for automated software testing, what we really need is automated test generation, over and above automated test execution.

Unfortunately most commercial automated testing solutions either only support automated test execution, or provide an environment to help write test-cases, which is quite different from automated test generation.

Explicit support for automated test generation is rare, if at all present. We now examine, at a high level, why this is the case.

What makes automated test generation hard?

Why Test1080?

Let us take a deep breath and examine the state-of-the-practice. Commercial offerings solve the “easy” automation of executing tests automatically, or automatically evaluating the coverage achieved by a test-suite. They do not give us the “hard” automation of automatically generating test-cases, for the sake of finding more errors.

Usually when we automatically generate test cases, there is some “goal” that needs to be met. This goal can be in the form of meeting a coverage criterion – generating tests which cover most of the program statements or program branches. Or the “goal” can be in the form of reaching certain specific target locations in the program, which the programmer views as dangerous locations.

Whatever be the goal in automated test generation, the inherent difficulty in automated test generation is a generic one. It requires **semantic analysis of the program**.

The term program analysis is also often overloaded or confused in certain commercial offerings. For example, I could analyse the program to look for variable names which contain the string “month”.   Presumably such an analysis which would involve a syntactic pass of the program text is not a deep one. For automated test generation, we need a semantic analysis of the program.

Any semantic analysis of the program involves an understanding of the underlying program representations. This starts from the underlying program representations in compilers capturing the program control flow such as the control flow graph. In addition, the flow of values need to be tracked during the very many possible program executions, so that the origin of an unexpected pernicious observed value can be detected. This is hard too. Most importantly, the process of test generation should be scalable – and work for both unit testing as well as system testing.

In our example program in this white paper, we saw one of the challenges in automated test generation – coming up with suitable input values which will expose program errors. This is only one of the challenges – and needs to be handled by analysis of the program control flow and data flow.

For automated test generation of mobile apps, there are additional challenges. A test input in a mobile app is a sequence of events, where each event may of the form of a tap or a touch. Each event furthermore has arguments, and the value of the arguments also need to be provided for constructing a single test input! Thus, for automated test generation – one needs to construct a sequence of events, and also the arguments of these events! Constructing a feasible sequence of events is also hard. Unlike conventional programming languages like C, Java – Android apps do not have a main method. So capturing the control flow is harder than constructing a control flow graph which is done by a modern compiler!

In summary, for automated test generation of mobile apps one needs to address challenges like (a) constructing a feasible sequence of events such as taps or touches (b) provide argument values to the events (c) produce such an event sequence and argument values such that certain errors in the app are exposed such as app crashes or hangs. Together, handling all of these challenges is hard. Furthermore, it needs to be accomplished efficiently so that the technique can work for any mobile app.

Test1080 is the answer to these challenges. It accomplishes automated test generation of Android apps. The generated tests can provide many value propositions such as exposing app crashes, battery drain, or meeting User Interface (UI) coverage. Most importantly it works for any Android app, a wide range of mobile devices, and is truly completely automatic.

With Test1080, the commercial world can have ***automated*** *testing for the first time*.

About the author

Abhik Roychoudhury is a Professor of Computer Science at School of Computing, National University of Singapore. His research has focused on *software testing and analysis*, software security, and trust-worthy software construction. He has been an ACM Distinguished Speaker (2013-19). He is currently leading the TSUNAMi center, a large five-year long targeted research effort funded by National Research Foundation in the domain of software security. He is also the Academic Director of the Singapore Cyber-security Consortium. He has authored a book on "Embedded Systems and Software Validation" published by Elsevier (Morgan Kaufmann) Systems-on-Silicon series in 2009, which has also been officially translated to Chinese by Tsinghua University Press. He has served in various capacities in the program committees and organizing committees of various conferences on software engineering, specifically serving as Program Chair of ACM International Symposium on Software Testing and Analysis (ISSTA) 2016 and General Chair of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE) 2022. He has been editorial board member of IEEE Transactions on Software Engineering (TSE) from 2014-18. Contact him abhik@comp.nus.edu.sg