Intelligently Choosing Testing Techniques

CS 390: Software Engineering

Dr. Hwang

By: Jonathan Bach

October 28, 2008
The ability for a company to produce a complicated, high quality, problem free product almost always requires the company to spend significant amounts of time and money in the testing of their product. While some companies have the resources to create gigantic test suites and use automated testing methods, there are several companies that do not have the funds or the time to test every single scenario. Dell’s website to customize a laptop purchase is a perfect example of a situation where producing an automated testing suite for all two million customer selectable combinations is not very practical. The reason why an automated testing suite would not be practical in this scenario is that it would take a single machine 64 hours to run the test suite while the customer selected customization options might possibly change within this time frame [4]. Some of the other testing techniques that a company could use instead of automated testing are random sampling, pair-wise testing, and N-wise testing. All of the various testing techniques have their own set of limitations and are practical in different situations. The ability to intelligently choose a set of testing techniques that can produce a product with an acceptable level of quality, without having to run a test for every conceivable combination or situation, is possible and practical.

The simplest testing technique that a company can use to validate the quality of their product is “random sampling”. The idea behind random sampling is that a company will eventually find the bugs if they start randomly checking different inputs. Random testing is fairly easy to implement and provides the company with a quick way to create a test suite [2]. When a company decides to use random sampling to test their product, they generally present their quality level in terms of a confidence level. If a company says that they are 99% confident that their product is 99% bug free then they are essentially guaranteeing that they pass the tests 100% of the time with a 1% error margin. In order to achieve a 99% confidence level with a 1% error margin for one million input combinations, a company would only need to run approximately 16,300 random sample tests [4]. In order to determine the sample size, a person would need to first calculate the critical value from the standard deviation of the 99% confidence level which is 2.58. Then a person would need to determine the percentage of picking a certain choice, which for a sampling size would be 50%. To calculate the total sampling size a person would multiply the critical
value by the picking percent, then divide by the error margin, and then square the answer. For the given situation of 99% confidence level with a 1% error margin, the maximum sample testing size for this software would be 16,641 tests. While random sample testing can greatly reduce the number of tests required to produce a high confidence level in the product, random sample testing never will be able to fully eliminate all of the bugs within a complex system. Even though random sampling tests can find certain types of bugs, random testing could also potentially cause an increase in the time it takes to find the actual location of the problem [2]. Due to the nature of tests being randomly placed throughout a program, it can be difficult to determine whether a failed test is the result of a computation right before the current test or the real problem is from a computation much earlier in the program that led to an incorrect result in the current computation used.

While random sampling will eventually uncover many and perhaps most bugs, the use of variations of random sampling that uses some knowledge of the system can allow the tests to uncover bugs more efficiently. Some testers use a variation of random sampling which puts more weight on tests related to an area in the system that changes the system’s behavior. Testing based on where the system’s behavior changes has a higher probability of finding bugs since there is a greater chance that a bug will occur during a behavior change or the bugs result in a different behavior. Another variation of random sampling is to break the input into equivalence classes or partitions and then perform random tests within those partitions. When a certain range of input is assigned to one equivalence class then each item in the given range while behave in the same manner and will produce the same output. A simple example of breaking a large range of input into equivalence class would be assigning grades. The full range of input is 0-100 but a person could easily break this range into six different equivalence classes. The six equivalence classes would be [invalid input (x<0, x>100)], [A (90-100)], [B (80-89)], [C (70-79)], [D (60-69)], and [F (0-60)]. Breaking the full range of inputs into equivalence partitions allows a person to only perform 6 tests to fully cover the entire input range. Equivalence partitions allows a tester to assume that if one test case discovers an error for a given input value within a partition then there would be other test cases that would find the same problem [3]. Therefore a tester would only need one type of test per
partition to detect a given error. In order to implement a test based on equivalence partitions, a tester would have to use knowledge of the input system and break the range of inputs into the proper equivalence partitions.

A combination of weighted random sampling and equivalence partition testing is known as “boundary condition testing”. A boundary condition test would be a test weighted towards the edges of equivalence partitions. While there are not specific rules for implementing the boundary condition tests on every type of input system, the general guidelines for the test cases are the beginning and ending to an input range and then two tests outside of the range on both ends [3]. Utilizing the understanding of a software system while implementing random sampling can greatly increase the probability of finding bugs while still covering the entire input range with a relatively small test suite size. Variations of random sampling are useful for finding bugs that are related to an individual input or single variables but have some difficulties when trying to identify bugs that are related to a combination of inputs or multiple variables.

In order for a company to increase the probability of finding the bugs related to a certain combination of inputs, it is suggested to use a “pair-wise testing” technique over a random sampling technique. The main idea behind pair-wise testing is to test every two variable combination without having to actually test every combination individually. A portion of the Dell system allows a user to select from three different categories like A, B, C. The three categories also allow the user to select three different options within the categories. In order to keep the example simple, let us assume that each of the options within a given category are a, b, and c. Since there are three categories with three options in each category, there are a total of twenty-seven combinations that a tester would have to run without using pair-wise testing. Instead of testing an individual combination of all three categories, pair wise testing allows a single test to break a single combination into two paired combinations which allows a tester to only need nine tests instead of twenty-seven. Let us assume that a given test could be represented as (<A>, <B>, <C>) where A, B, and C are the three categories and <A>, <B>, and <C> would be some option from A, B, and C. The nine tests that a tester would perform are (a, a, a), (b, b, b), (c, c, c), (a, b,
c), (b, c, a), (c, a, b), (a, c, b), (b, a, c), and (c, b, a). The first test, (a, a, a), would test the combination of (a, a, -) and the combination of (-, a, a) which then the nine tests covers every single three combination input choice. With the Dell system that allows customizing a laptop pc there are 2,000,000 unique combinations of inputs that would only require 36 pair-wise tests to completely test these combinations [4]. As the number of input combinations grows, the reduced number of tests becomes more evident. Pair-wise testing has become popular with configuration systems since they are easy to manage and execute, reduces the number of tests, and is believed to catch some of the more important bugs that could occur [1]. This idea of pair-wise testing a system can easily be expanded into a more general strategy known as “N-wise testing”.

N-wise testing is a testing technique that tests based on a combination of N number of inputs. Pair-wise testing is equivalent to N-wise testing when N happens to be two. Studies have show that testing three inputs covers approximately 90% of the system. Testing only three input combinations might be more practical than testing four input combinations because testing four combinations requires more tests that might not even uncover new bugs that the three input combinations tests can already detect [4]. N-wise testing could be beneficial when testing a security system by continuously checking unexpected or invalid input changes that could be caused by a security attack [2]. While N-wise testing seems to be a decent choice when dealing with combinations of inputs, N-wise testing makes a poor assumption that the categories within the combinations are independent of each other [2, 3]. Additional tests would need to be added to the test suite in order to test the various ordered input sequences [4].

While random sampling and N-wise testing are techniques that a company can easily implement in order to test a system while keeping the test suite size down, another technique suggests putting some careful thought into how the system is designed and then to test the system based on dependencies found to exist within the system. Random sampling and N-wise testing allows a company to test a system without the need to have any type of knowledge of how the system works. After a company spends sufficient time looking at the interdependencies within the system, usually the complex system is found to actually be composed of multiple independent systems and smaller systems that are connected through a
single control [4]. When a complex system gets broken down into two or more completely independent systems, then the test suites created for the individual systems will be smaller than for the complex system because a N-wise test that is a combination of the two systems would not need to be tested. If a complex system were to get broken down into two systems that depend on a common control, then a company could weight random testing towards the common control which would allow the company to uncover the system’s bugs more efficiently.

Even though dependency testing can produce smaller test suites for complex systems, the ability to reason through input requirements can also help reduce the number of tests in a test suite. If it is possible to logically combine two options in an input category, then it would help reduce the number of N-wise tests required in a test suite. Suppose that a system has three states that are based on three different types of shipment orders. While the system is only required to contain three types of shipment order, the company might want to break the type of shipment orders up into subtypes to allow smaller and more specific groupings of the shipment orders. Even though this would cause more combinations for a N-wise test to check the combinations between the different order types, the company could choose to only test the combinations between the three main order types and the three states since that meets the requirements of the system [4]. The ability to reason through requirements is important when a company is trying to keep the test suite size small and does not need any meaningless or redundant tests.

There are several testing techniques that can help reduce the number of tests required to fully cover a complex software system. The easiest test technique to implement and which requires no knowledge of the software system is “random sampling”. Even though random sampling will eventually find the bugs within a system, using knowledge of the system to implement a variation of random sampling can greatly increase the probability of finding the bugs allowing the tests to be more efficient. When a system deals with combinations of multiple inputs, then utilizing a “pair-wise testing” technique can significantly decrease the number of tests require to cover the inputs selections. After understanding how pair-wise testing is implemented, a tester could easily generalize the strategy into “N-wise testing” and further reduce the number of tests required to cover a larger number of input combinations. The
testing technique that requires an extensive amount of knowledge about the software system is “dependency testing”. Dependency testing is extremely effective because it allows a tester to focus the tests onto a few controls that the entire system relies on and where the most vital bugs would be located. Dependency testing also reduces the number of tests required in a test suite and helps prevent unneeded and redundant tests to be placed into the test suite. All of the above testing techniques allow testing to become practical for companies since they all significantly reduce the test suite size and often cause the time required to run tests to decrease substantially. While running automated tests for every situation and using testing techniques that do not require knowledge of the software system are possible, it is not practical for a company to spend the time and money on these tests because by utilizing some knowledge of the system a tester can easily develop tests that are more efficient at finding bugs and thus require less time and money.
References


