A Permutation Technique for Test Case Prioritization in a Black-box Environment

Lucas Albertins de Lima, Juliano Iyoda and Augusto Sampaio[1]

[1]Centro de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851 – CEP 50732-970
Recife – PE – Brazil

{lal2,jmi,acas}@cin.ufpe.br

Abstract. *Test case prioritization techniques can be used as an effective way to reduce the test execution time. Most of the prioritization techniques are related to regression testing and assume that the source code is available. Few of them address the problem at the initial testing phase (non-regression) or in a black-box testing environment. In this paper, we present a technique based on permutation generation to order a set of test cases aiming to reduce the test execution time and, consequently, increase productivity. This work is developed in the context of a collaboration project with Motorola BTC (Brazil Test Center), where we show the feasibility of our approach applied to some of their (black-box) test suites. A prototype which automates our test sequence generation is used to illustrate our approach in a case study, which has produced encouraging results so far.*

1. Introduction

Software testing can be an arduous and an expensive process [Beizer 1990]. Companies have put a lot of effort on this activity, which has shown to reduce costs and the time-to-market of their products. Therefore, any effort to optimize the creation and the execution of test cases helps companies to survive in the current competitive business environment. Test case prioritization is an approach to optimize the execution of tests by ordering a set of test cases. Different prioritization techniques use different ordering criteria: execution time reduction, early discovery of defects, faster test coverage rate, etc.

In this paper we present a strategy to prioritize test cases based on permutation generation to minimize the time spent over the test execution. Our approach was developed in a black-box test environment of cell phone devices as part of a cooperation project with Motorola BTC. We reduce the total execution time by reducing the time spent to *configure* a cell phone from one test to another. Although permutation cannot be applied to very large test suites, whenever the test suite's size does not exceed 17 tests, permutation is not only feasible but also generates the best sequence available.

In this project we find many test suites of sizes small enough to apply our approach. We carried out a case study which shows an improvement of up to 30% in the configuration execution time in comparison to a sequence generated manually based on heuristics.

There are various works on test case prioritization [Rothermel et al. 2001, Elbaum et al. 2002, Jeffrey and Gupta 2008, Elbaum et al. 2000, Srikanth 2004,

Qu et al. 2007]. Most of them are applied to a white-box environment. All of them try to find sequences of test cases which bring gains with respect to specific objectives, but none of them finds the best sequence as they use heuristics. As we propose a brute force permutation algorithm that always finds the best solution, we regard our approach as a complementary technique to those previous works. Differently from them, our main concern is not to find good heuristics for sequence generation (as we generate all of them), but to find good metrics which accurately capture a test suite execution time.

The remainder of this paper is organized as follows. Section 2 presents our approach and briefly describes a prototype which automates the test sequence generation. A case study which illustrates our approach is presented in Section 3. Section 4 describes related work and Section 5 concludes.

2. A Permutation Technique for Test Case Prioritization

We consider only functional tests generated from requirements documents. The source code of the system under test is not available to us. The test cases are specified in English and are executed manually.

Our ordering aims at reducing the time spent on test execution. In order to achieve that, we use the idea of reusing the configurations (the state) left in the cell phone by a test [Rodrigues et al. 2007]. For example, suppose that one of the steps of the test T_1 stores a new contact in the phonebook. If there is another test, say T_2, whose precondition is the existence of a contact in the phonebook, then such test could simply run after T_1. Otherwise, the tester would have to add this contact to configure the phone *before* executing T_2. This ordering saves time and effort from the tester as the new contact is already in the phonebook as a byproduct of T_1.

The prioritization problem has been formally defined as follows:

Definition 1 *Let S be a test suite, P the set of permutations of S and f a function from P to the real numbers. The test case prioritization problem consists in finding $p \in P$ such that*

$$\forall p' \in P.\ (p \neq p') \Rightarrow (f(p) \leq f(p'))$$

The function f represents a measure function which allows us to compare two permutations.

The main problem of test cases prioritization is the generation of P. We have to produce all permutations of test cases, which in practice is not feasible for a large number of tests. For instance, assuming that it takes 1μsec to generate a single permutation, the permutation of up to 12 tests takes no more than few minutes. But, for 17 tests, it takes 10 years. For more than 25 elements, the time required is far greater than the age of the earth [Sedgewick 1977]. Because of this, all related works focus on the usage of heuristics in the prioritization problem. However, we noticed that several test suites used by Motorola BTC contains less than 17 test cases. This was the main motivation to develop an approach which exhaustively looks for the best sequence. Therefore, unlike all related work, we are not concerned with the definition of appropriate heuristics for test sequence generation. Our main concern in this work is the definition of an appropriate f which captures the correct notion of an efficient sequence. In our case, f is related to the level of reuse of phone configurations that a sequence possesses. Our function f

computes the time the tester spends configuring the phone between the execution of two tests.

Before detailing the permutation strategy, we define some concepts related to a test case:

Identifying Information ID, suite and description.

Setups A set of configurations that does not change during test execution of a suite. For instance, the set up of specific configuration bits on the cell phone does not change during the execution of the entire suite. These configurations should be executed before the whole test sequence.

Phones Each phone can be a PUT (Phone Under Test) or an auxiliary phone.

Input/Output Input is the required state of the phone before the execution of a test. Output is the state of the phone after the execution of a test. The input is data required to be stored on the phone before the execution of a particular test. The output data represents the state of the phone when the test is finished. For instance, "the phone contains 3 GIF files" and "there is 1 contact in the phonebook" are examples of inputs and outputs.

2.1. The Permutation Technique

We develop a technique based on permutation generation of all possible sequences from a test case set. This decision is justified by the fact that many test suites we deal with have a limited size of elements, which allows computers of today to generate all possibilities and return the best results.

Based on this idea, we developed an algorithm that generates the permutations with a pruning mechanism for optimization. During the generation of a particular permutation, the algorithm checks if the (partial) sequence generated is already worse than the best sequence found so far. If so, it does not generate the rest of that particular subsequence, nor any sequence with that specific prefix, as the time can only increase by adding more tests to the sequence. We use the term *cost* to refer to the time taken to configure a phone.

Our function f reflects the level of reuse of configuration data (input and output) of cell phones. The cost computed by f is simply the total time spent with *configurations* after executing the entire test suite. Figure 1 shows a simplified example of how the calculation works. Each box represents a test case, where we only show its input and output data. On the first element, we cannot reuse any data as there is no previous test. So, we add to the total cost of the sequence the time to add the input data of the test A. In this case, we add the time a tester takes to insert 1 Email. The time to setup each kind of configuration was measured in an experiment. After the test A is executed (the steps for executing the test are omitted from the Figure 1), we can reuse the outputs of A in the inputs of B. Note that the execution of test A produced a second Email. So, before executing the test B, the tester does not need to add the two Emails required, only the GIF file. The time spent on adding a GIF is added to the total cost. At last, from B to C, we can reuse the two Emails and one of the two GIFs needed in C, adding to the total cost only the time spent on adding one more GIF.

There is still one issue missing in this calculation: what about tests with multiple phones (say, several PUTs and several auxiliary phones)? How do we reuse their input-output information? This scenario makes the calculation a little bit more complex. This

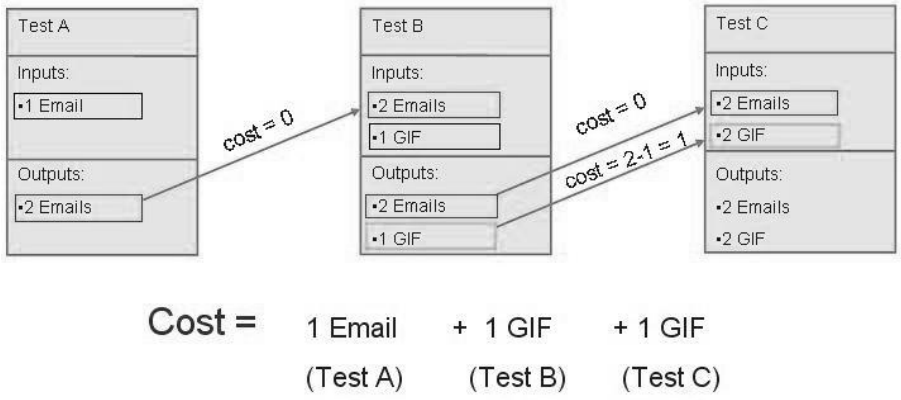

Figure 1. An example using the reuse function.

happens because, before calculating the final cost, we have to combine, say, each PUT's output with another PUT's input to check which combination produces the lowest cost. Figure 2 shows a possible combination where the outputs of the phone X are reused by the phone Y′. Similarly, the outputs of phone Y are reused by the phone X′ (assuming that all phones are, say, PUTs).

To perform this verification, we apply an algorithm similar to the permutation one. For instance, the phones of test A in Figure 2 are combined with phones of the same kind in test B. By traversing these matchings we can verify which pairs make more reuses. As tests usually require one or two phones (three at most), the generation of these pairs does not impact the total sequence generation time. We do not promote reuse from a PUT to an auxiliary phone (or vice-versa) as, in general, auxiliary phones cannot be used as PUTs.

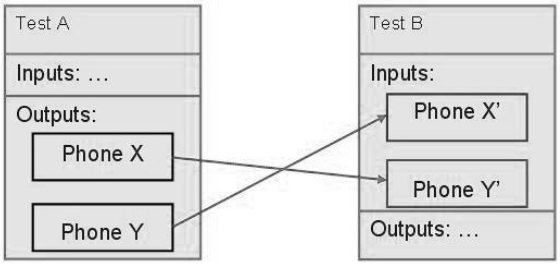

Figure 2. A possible phone combination.

Another aspect related to the calculation of the cost is the decision to accumulate the state of the phones from one test to the next. Keeping the state means that a phone that is used through a sequence of tests can accumulate data from the reuses of all test cases executed. An example can be seen in Figure 3, where the state of test A is updated with an Email and a GIF file. Note that the GIF file is not used by the test B, but it is still in the phone state. So, it can be reused by the test C without any configuration cost added to the sequence. The GIF file is only reused by test C, as a result of the accumulated state of the phone after executing tests A and B.

Alternatively, we can calculate the cost without taking into account the cumulative phone state by looking strictly at the possible reuses between two test cases. In this case, the cost of running two particular tests is immutable regardless the sequence (the

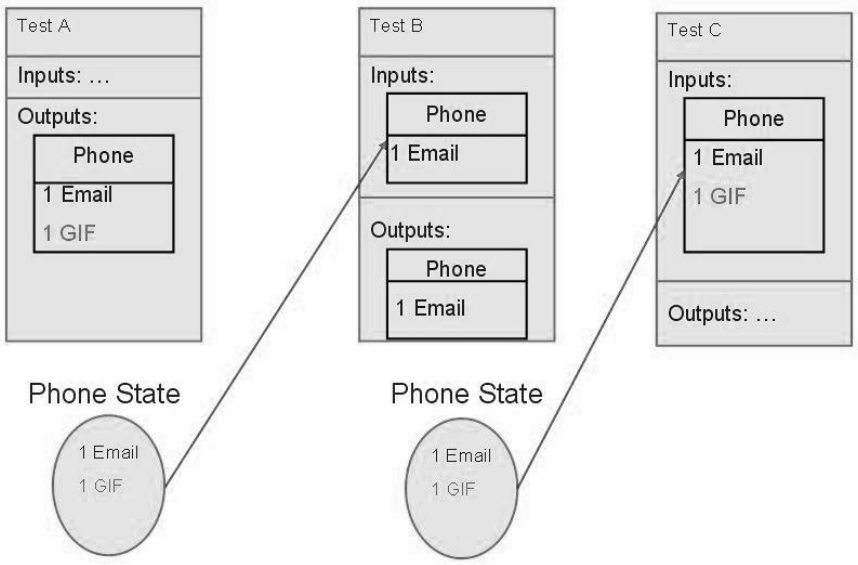

Figure 3. An example keeping the phone state.

context) they belong to. This happens because they do not suffer any interference from the execution of any other tests executed previously. This variation allows us to calculate the costs of reuse between any two tests in advance before generating the permutations.

Both stateless and stateful approaches have advantages and disadvantages. The former has a better performance as the costs between tests do not change and their costs are calculated only once. Therefore, they can be stored in advance and inspected when needed. This cannot be applied to the stateful algorithm as the state of the phones changes along the sequence and affects all the following tests. So, each possible sequence requires the cost to be computed on the fly. On the other hand, the stateful approach undoubtedly produces a more accurate calculation concerning the execution time.

2.2. The Tool

A tool has been developed to allow us to run some experiments to validate our approach. Both stateless and stateful approaches have been implemented. The tool allows the user to store test cases and their attributes (ID, suite, description, setups, input and output).

After including the test cases, the user can choose a group of them to be prioritized, as shown in Figure 4. By pushing the *prioritize* button a specific permutation algorithm is run (stateless or stateful).

The best sequences are shown in a table. Notice that depending on the amount of reuses, the number of best sequences can be very large (several sequences might have the same lowest cost). The fewer the reuses among the tests, the larger the set of best sequences. This happens because the number of possibilities is huge. For example, for 12 tests, there are 479,001,600 sequences, so there is a high probability of existing more than one best sequence. The tool also displays the expected time spent on data configuration between tests. Notice that this time does not correspond to the whole sequence execution time, but only to the sum of all configuration times. The time spent on the execution of the test steps is not part of our cost calculation.

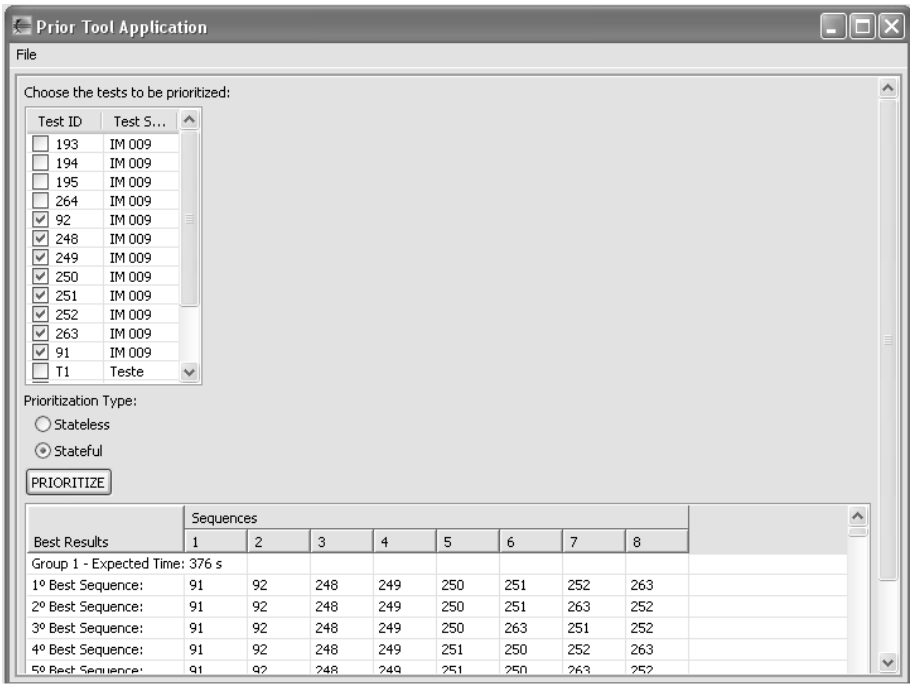

Figure 4. The prioritization tool.

Table 1. Comparison among approaches.

Generation Type	Generation Time	# best results	Configuration Time
Manual(Stateless)	Around 2 weeks	1	641
Manual(Stateful)	Around 2 weeks	1	617
Automatic(Stateless)	53 sec	96	455
Automatic(Stateful)	7 min 21 sec	45360	431

3. Case Study

We used the tool presented above to carry out a case study in collaboration with the Integration Test Team of Motorola BTC. This team already uses heuristics to generate test sequences manually [Rodrigues et al. 2007]. Their approach has already brought some gains to their performance. In this case study, we work with an specific software component test suite. It contains 12 test cases with all configuration data available.

Before carrying out the case study, we collected the execution time of each kind of configuration in a small experiment. We measured how long does it take to, say, add a contact in the phonebook, add a GIF file to the cell phone, delete an Email, etc. We measured these times from two testers: a beginner and an expert. They performed the setup of all kinds of configurations (27 in total) in different sequences. These times were used to calculate the configuration time of the test sequences generated by both the manual approach and by our tool.

The comparison between these approaches is shown in Table 1. In the first two rows we see that the manual methodology takes around two weeks to generate the sequences. This happens because the testers devote a small period of time per day to work on the sequence generation. So, the whole process can drag on for two weeks. The manual

approach uses a heuristic which always returns a single sequence, which is not guaranteed to be the best one. The last two rows show the outcome of our tool. Not surprisingly, its generation time is far better than the manual generation. The large number of draws returned by our tool was explained above (the large number of possible sequences and few opportunities for reuse can impact the amount of best sequences). In the case of the stateful permutation, this number is even larger. This happens because a particular test placed at a specific position in the sequence may well be placed in several subsequent positions with the same cost (provided that the subsequent tests do not delete configurations). This increases the number of possible best sequences in comparison to the stateless permutation.

The configuration time on both stateless and stateful techniques improved the *configuration* execution time in around 30% in comparison to the manual approach (i.e., the time spent to execute the *configurations* of the sequences generated by the tool is 30% less than the time spent to execute the *configurations* of the sequence generated manually). This is not surprising as the tool verifies all possible sequences and therefore guarantees the best results. Also, a sequence created manually has high probability of not been the best one because some group of reuses can be difficult to visualize by a human being, especially if we think of the amount of possible permutations. Hence, providing an automatic generator is doubly advantageous: it produces the best execution sequences in less time.

4. Related Work

This section briefly describes related works on test case prioritization and illustrates the variety of criteria applied to test sequence generation.

Rothermel et al. show empirical results from code-based techniques applying different strategies like coverage of statements and branches [Rothermel et al. 2001]. Their main aim is to improve the rate of fault detection measured in terms of APFD (Average of the Percentage of Faults Detected). This approach was extended later to provide more techniques using coverage of functions [Elbaum et al. 2002]. Jones and Harrold presented a technique which prioritizes test cases using the modified condition/decision coverage (MC/DC) criterion, which is a form of exhaustive testing that uses branch coverage [Jones and Harrold 2001]. Srivastava and Thiagarajan propose a test case prioritization technique based on basic block coverage for large systems [Srivastava and Thiagarajan 2002]. Kim and Porter modeled regression testing as an ordered sequence of tests and presented a history based on a prioritization technique which utilizes information from previous tests [Kim and Porter 2002]. Jeffrey and Gupta propose techniques based on statements coverage that takes in consideration how statements affect each other [Jeffrey and Gupta 2008]. Qu et al. present a technique applied to black-box testing but still in the regression test context [Qu et al. 2007]. It uses historical information of test cases execution and classifies them based on the fault types they revealed. Avritzer and Weyuker propose test case prioritization to other test phases different from the regression test [Avritzer and Weyuker 1995]. They present techniques to order test cases of systems modeled as Markov Chains. Shrikanth proposes an approach to prioritize test cases using requirements engineering research to identify severe faults earlier [Srikanth 2004].

The approach proposed by Rodrigues et al. is the main motivator of our work [Rodrigues et al. 2007]. They propose a methodology to construct sequences of tests to be executed in a ordered way on the context of cell phone testing at Motorola BTC. They use a tree structure where each node represents a test, which is modeled as a pair of input and output data. The root node is a test which requires no initial configuration effort for its execution. The following nodes are those whose inputs intersect the outputs of the root node. Such nodes reuse the current state of the cell phone. The remaining tree is built using the same criterion. Once the tree is constructed, the longest path from the root to a leaf is considered the best sequence to run. This approach, as ours, is based on the reuse of configurations of the phone, which reduce re-work and, consequently, the effort spent on the execution. Despite their good results in comparison to an ad-hoc order, the heuristics proposed do not guarantee the discovery of the best sequences. For example, a sequence which requires more effort initially but reuse a lot more at the end can be better overall than a sequence which, although begins with no effort, takes no advantage of reuse. For a large test suite, however, this technique provides a better (feasible) alternative in comparison to permutation. The same analysis is valid with respect to all related work described above.

5. Conclusion and future work

In this paper, we described a new technique for test case prioritization in a black-box environment. Our technique applies permutation generation to obtain test sequences which maximizes the reuse of the phone state from one test to the next.

Our algorithm takes into account the time to configure the phone between tests. The time to setup each kind of configuration was measured in a small experiment with two testers. We developed a tool which mechanizes our algorithm and which has been used to perform a case study. Our preliminary case study gave us encouraging results which showed that the best sequences could reduce the *configuration* execution time in up to 30% in comparison to a sequence created manually.

Contrary to all related works, we have adopted a brute force approach to test sequence generation. In the context of our collaboration with Motorola BTC, we found that many test suites contain less than 17 test cases. In these cases, the generation of the best sequence is feasible. Therefore, our experience shows that we cannot always neglect brute force approaches. We are not claiming that brute force is always superior than the approaches based on heuristics. We present a *complementary approach* to the existing works which performs better whenever the suite size restrictions are respected.

The technique presented can be generalized to any stateful system, mainly when configuration data are critical. For instance, in systems using databases like a video rental software, a test to remove a client should run after a test that adds a client. This order saves the time spent to add a client to the database before testing its deletion. Although the main concept of state reuse can be generalized, the actual implementation of the cost function may vary in each application domain.

For future work, we plan to carry out some experiments to measure the execution time of our sequences in comparison to those created manually. This experiment is to confirm that the configuration time do impact the total time of the test execution and also to provide a detailed comparison between our approach and Motorola BTC's manual

technique [Rodrigues et al. 2007].

Another issue to address is the reduction of the number of best sequences generated. To minimize this, we intend to implement an algorithm that detects transitive closures to find tests which do not depend and do not influence any other. These tests can be removed from the permutation. Another optimization is to create equivalence classes of tests which have specific configurations such that one test can represent all others in the permutation algorithm.

We also intend to implement user-defined clusters inside a test suite. This means that the user will be able to choose subsets (clusters) of tests from the test suite which will be ordered first. Afterwards the clusters are prioritized with the remaining tests and other clusters. Each cluster is then treated as single test. To do this, we simply consider the inputs of the cluster to be the inputs of its first test and, consequently, the outputs of the cluster to be the outputs of its last test. This functionality is useful when external factors not related to configuration reuse can interfere in the generation of the best sequences. This feature will improve the scalability of our approach by allowing large test suites to be ordered. However, the best sequence may not be generated anymore. In addition to this feature, we intend to investigate other heuristics to generate test sequences from large test suites.

References

Avritzer, A. and Weyuker, E. J. (1995). The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. volume 21, pages 705–716. IEEE Transactions on Software Engineering.

Beizer, B. (1990). *Software Testing Techniques*. International Thomson Computer Press.

Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2000). Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software*, volume 25, pages 102 – 112. ACM.

Elbaum, S. G., Malishevsky, A. G., and Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *Software Engineering*, 28(2):159–182.

Jeffrey, D. and Gupta, N. (2008). Experiments with test case prioritization using relevant slices. *J. Syst. Softw.*, 81(2):196–221.

Jones, J. A. and Harrold, M. J. (2001). Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, page 92.

Kim, J.-M. and Porter, A. (2002). A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA. ACM.

Qu, B., Nie, C., Xu, B., and Zhang, X. (2007). Test case prioritization for black box testing. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 465–474, Washington, DC, USA. IEEE Computer Society.

Rodrigues, H., Baden, R., and Júnior, W. (2007). A Test Execution Sequence Study for Performance Optimization. Technical report, Federal University of Pernambuco.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *Software Engineering*, 27(10):929–948.

Sedgewick, R. (1977). Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164.

Srikanth, H. (2004). Requirements Based Test Case Prioritization. In *Student Research Forum in 12th ACM SIGSOFT Intl Symposium*. ACM SIGSOFT, on the Foundations of Software Engg.

Srivastava, A. and Thiagarajan, J. (2002). Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106.