TRANSFORMING AUTOMATED SOFTWARE FUNCTIONAL TESTS FOR PERFORMANCE AND LOAD TESTING

Peter Sabev[1], Katalina Grigorova[2]
[1]PhD Student, "Angel Kanchev" University of Ruse, Ruse, Bulgaria
[2]Associate Professor, "Angel Kanchev" University of Ruse, Ruse, Bulgaria

ABSTRACT

Functional testing is a quality assurance process that could be automated. Functions are tested by feeding them input and examining the output expecting concrete results. However, the performance measurement and execution times of these tests are rarely considered when executing functional tests. Adding simple timestamps for every functional test execution when such a test is started or stopped could bring a valuable benchmarking data for analysis, save a significant amount of time for executing performance tests separately from functional tests, and increase defect removal efficiency.

Keywords: *Software Engineering, Software Testing, Testing Model, Automated Testing, Functional Testing, Performance Testing, Regression Testing, Benchmarking*

INTRODUCTION

In a typical programming project, approximately 50 percent of the elapsed time and more than 50 percent of the total cost are spent in testing the developed program or system [3]. Software testing has been the main form of defect removal since software began more than 60 years ago. There are at least 20 different forms of testing, and typically between 3 and 12 forms of testing will be used on almost every software application [1].

Functional (or black-box) testing has become one of the most popular testing methods – it can be easily designed, executed and implemented for automatic regression tests. This method verifies correct handling of the functions provided or supported by the software, or whether the observed behavior conforms to user expectations or product specifications by only provisioning simple input data and comparing the returned data (i.e. the actual results) to the expected results.

However, using only functional testing seems to be 35 percent less efficient, i.e., it finds only about one bug out of three [1]. That is why alternatives that combine higher efficiency levels with lower costs are worth considering. Benchmarking the software using performance testing and analysis is such an important activity. It provides key metrics such as response times or throughput rates under certain workload and configuration conditions.

CONTEMPORARY STATE OF THE PROBLEM

Since performance analysis is not always part of software engineering or computer science, many software engineers are not qualified to deal with optimizing performance [1]. Although these measurements are important, they are rarely performed during testing in many companies and projects. Even the most mature test processes divide the functional and performance testing in separate activities and separate subprojects done by completely different specialists.

One of the reasons behind the strong separation of performance and functional test results is that the widely adopted IEEE 829-2008 - IEEE Standard for Software and System Test Documentation sets items pass/fail criteria [4] that is often mistaken by the majority of IT specialists as functional test pass/fail.

Although passing a test means that software has met its functional and non-functional requirements (such as expected response or processing time), software testers rarely pay attention to the performance measurements during test execution; passing or failing used to be enough during the years and additional performance tests were done if needed.

All widely known modern test result formats provide ways to store test execution time (xUnit has <testcase time="x">, TestNG has <test-method started-at="yyyy-dd-mmThh:mm:ssZ" finished-at=" yyyy-dd-mmThh:mm:ssZ ">, JSON Test Results format has float variable called time, any log file has timestamps and any custom XML/HTML file can have relevant time-related tags). However, the majority of test execution management systems and bug tracking systems consider tests output as pass/fail flag only, so even if recorded, functional tests time measurements usually remain neglected. Nowadays, even if the testers want to log performance measurements they will eventually need significant modification in the default software configuration. A research based on [6], [7] and [8] was made for 48 of the most popular software test management tools (Aqua, Assembla, Bstriker, Bugzilla, codeBeamer, Enterprise Tester, Gemini, HP Quality Center, IBM Rational Quality Manager, informUp Test Case Management, JIRA, Klaros-Testmanagement, Mantis, Meliora Testlab, Occygen, Overlook, PractiTest, QAComplete, QABook, qaManager, QMetry, qTest, RADI, Rainforest QA, RTH-Turbo, Silk Central Test Manager, Sitechco, Tarantula, TCW, tematoo, Test Collab, TestComplete, TestCube, Testersuite, Testitool, TestLink, TestLodge, Testmaster, Testopia, TestPad, TestRail, TestTrack, Testuff, TOSCA Testsuite, WebTST, XORICON TestLab, XQual, XStudio and Zephyr). As a result, all of the tools provide test pass/fail reports and none of them has integrated ready-to-use automated reports for functional tests with build-to-build comparison: this is either not supported at all, or requires additional plugins, customization or separate setup for a dedicated performance testing tool or feature.

However, many functional tests are automated in everyday life. This is performed by repeatedly executing these tests, which avoids human mistakes during execution and ensures faster retrieval of results. Such automated functional tests could be slightly modified to collect performance data by using several checkpoints and collecting timestamps before and after each test case or even before test step execution.

According to Table 5-6 (Defect Removal Efficiency by Defect Type) in [1], adding performance measurements to automated functional tests could improve the defect removal effectiveness by 5% for requirement-related defects, 10% for design-related defects and 70% for performance-related defects.

SEQUENCE DIAGRAM OF A TYPICAL AUTOMATED FUNCTIONAL TEST

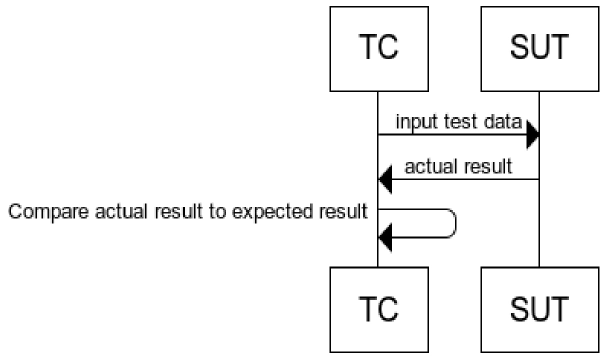

Figure 1. A typical functional test

A typical automation functional test is performed by using a test controller (TC), an external application or module that is able to independently configure, execute and terminate the software under test (SUT) directly or by using one or more test agents. Figure 1 shows a typical time sequence diagram for simple automated functional test.

ADDING CHECKPOINTS WITH TIMESTAMPS TO THE FUNCTIONAL TEST

The first step of the suggested approach is to modify the time controller by adding two checkpoints – one at the beginning (immediately before the test execution has started) and one at the end (immediately after the test execution has ended) as shown on Figure 2. Thus, one could easily calculate the execution time as the time difference between the two checkpoints. Note that the modification is done only in TC, and SUT remains unchanged.

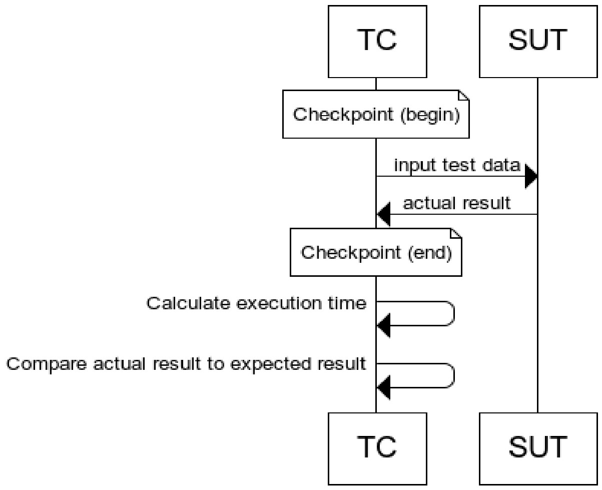

Figure 2. A measurement before and after functional test execution

In this case it is possible any combination of several different tests or same tests to be run on different systems, subsystems, features, modules, units, versions or environments; it is only required to follow the practice and bracket each test case run with time measurements as shown in Figure 3.

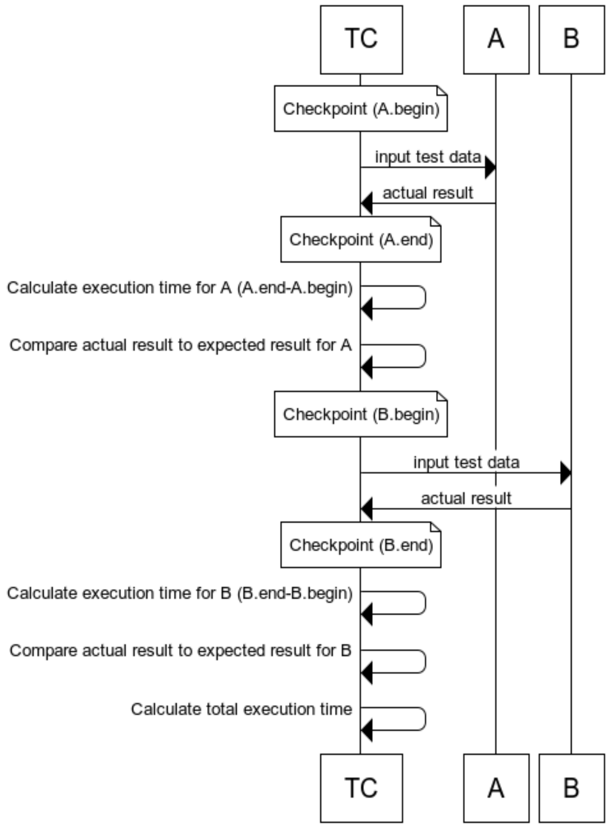
TC
A
B
Checkpoint (A.begin)
input test data
actual result
Checkpoint (A.end)
Calculate execution time for A (A.end-A.begin)
Compare actual result to expected result for A
Checkpoint (B.begin)
input test data
actual result
Checkpoint (B.end)
Calculate execution time for B (B.end-B.begin)
Compare actual result to expected result for B
Calculate total execution time
TC
A
B

Figure 3. Executing several functional tests with time measurements

Note that two metrics for total execution time can be obtained depending on what measures are needed:

1. Total elapsed time during test execution: (B.end-A.start)
2. Total test execution time: (A.end-A.start) + (B.end-B.start) + …

However, the nature of performance measurements has a significant limitation – to consider functional test performance results as valid, the functional test should have passed. In a case where performance drops to zero when a high-severity bug is encountered and stops the test from running properly, test results should be ignored [1].

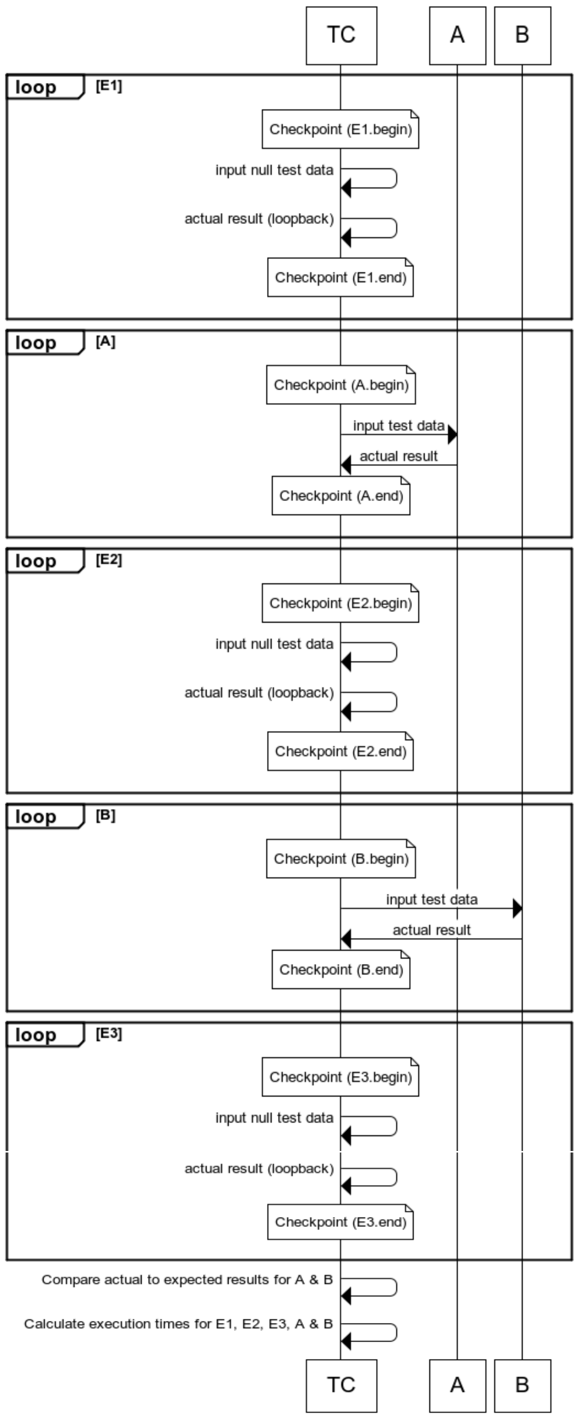

Figure 4. Final state after modifying tests for performance measurements

FINE TUNING THE PERFORMANCE MEASUREMENTS

Until now functional tests served as baseline tests for one single transaction in isolation as a single user. The proposed methodology allows ramping up to a desired target maximum currency or throughput for the transaction by executing same tests several times. This could be either synchronous or asynchronous, using one or multiple test agents covering one or multiple test scenarios; the possible combinations are limitless and the only requirement is to repeat the tests and their time measurements a certain number of times. The more measurement points and iterations are performed, the more time for execution is needed; however, a better precision of the performance measurements is obtained [5].

If tests fail or any other problems are encountered at this stage, one only needs to run isolation tests to identify and deal with what’s wrong. This would be followed by a load test combining all transactions up to target concurrency and then by further isolation tests, if problems are discovered [2].

Although modifying the SUT is not required to apply the benchmarking method proposed here, there are a number of performance tools and measurement devices such as profilers that collect data on the fly. It is also possible to embed performance measurement capabilities into software applications themselves, which is called instrumentation. Since instrumentation and other forms of performance analysis may slow down application speed, one must take care to ensure that the data is correct [1].

When executing the modified tests, it is important to remove (or at least limit) any external factors that could affect the performance measurement results. Typically, these include inconsistent network bandwidth usage, current user load, noises or signal disturbances, different database states (all of these are especially valid when testing remote or third-party systems), resource allocation as CPU, disk or memory used by other processes or users (all of these are especially valid when using virtual machines), TC consumes a lot of time or resources compared to the single test execution (especially valid for simple, fast tests), caching, etc. Normal user flow should also be considered and user response time must be added to functional tests, so baseline tests to establish "ideal" response-time performance could be performed [2].

In common practice it is required to benchmark the "pure" response time for a given SUT feature, module, unit or process. That is why additional empty tests could be run at the beginning and at the end of the test execution, as well as between every other test measurements. These tests could allow measuring average time for specific test maintenance operation such as pausing in the system, resetting environment, setting specific configuration, reverting snapshot, database restore or cache delete.

Even for the simplest test, it is a good idea to bracket it for an "empty" test with null data. For example, if a web application is tested with an online JSON query, variations of such "empty" tests could be times for: localhost ping, sending/receiving request on localhost, sending/receiving empty request on localhost, sending request second time in order to obtain the response via proxy or server cache, etc.

Another important point is to minimise any activity that could impact the performance and that could be avoided at that moment (e.g. writing in logs, time calculations or anything else that could be postponed). Thus, it is advisable all calculations performed in the middle of the test execution (Figure 3) to be postponed after the benchmarking finishes (as shown in Figure 4).

Considering all the tests have passed and the time measurements are correct, the latter could be amended accordingly with the "empty" test results. In the example above, if the average time for server response to empty JSON query is 0.5 seconds and a typical baseline JSON query is 3.7 seconds, obviously the "pure" JSON query processing time is 3.7 – 0.5 = 3.2 seconds. The exact calculations and amendment of actual test results are highly dependent on the testing context and should be reviewed individually according to the project and test case specifics.

Figure 4 shows the final sequence diagram with all the fine tuning proposals implemented.

Calculating minimum, maximum, and average times allows even deeper look into stability, time execution margins, and finally, early diagnosis of potential software defects. Once the performance tests data measurements are collected and analyzed, they can be used as valuable baseline against future releases, comparison between two software implementations, etc.

CONCLUSIONS AND FUTURE WORK

This article presents a novel approach to functional test design that enables collecting important performance and benchmarking data by bracketing each functional test execution with timestamps.

This work differs from existing approaches in that it allows partial performance and load testing to be done while executing functional tests, with minor additional effort. There are several benefits to this method: it allows early detection of performance defects, and potentially increases defect removal efficiency.

The theoretical ideas presented in this article have been successfully applied in the real-world in three different software companies: Experian Decision Analytics (in 2010, during the testing of their application fraud prevention software called Hunter), Digital Property Group (in 2012, during the testing of company's real estate websites), ProSyst Labs (in 2014, during the testing of J2ME hardware devices for a Deutsche Telekom's smart home project called QIVICON) and Certivox Ltd. UK (in 2015, during the testing of M-Pin zero-password authentication software product). Continuous performance benchmarking data obtained from the functional tests allowed build-to-build comparisons, which assisted the early identification of several defects with critical severity. One of the recent examples for the effectiveness of the proposed approach in practice is illustrated on Figure 5. An automated functional test case consisting of simple user login and logout is triggered and executed every night at 1:00h. Test execution usually took between 3.6 and 5.0 seconds. A recent code change on 09-May-2015 introduced a performance issue that caused the total test execution time to be above 6 seconds which was confirmed during the next executions during the weekend. On Monday, 11-May-2015 the code change was reverted and further performance investigation were made during the next days. As a result, performance improvements were done and the execution time dropped under 1.7 seconds.

However, there are some practical and theoretical issues that need to be addressed regarding this approach. On the practical side, the proposed method for obtaining benchmarking results by modifying functional software tests cannot fully substitute regular performance, load or stress testing activities. Even in the case when such substitution is theoretically possible, it would be impractical due to the complexity of implementing and maintaining different multi-agent functional test execution scenarios. Another issue is the time correction using the “empty” tests described above, which needs to be further researched and improved for further test results.

Much remains to be done in this regard, especially when precise performance data is required and many test iterations are not a practical workaround solution. However, if the final goal is just obtaining some performance data that could serve as initial indicator of SUT performance, then this is a relatively easy approach, which has potential for very good return of investment. This work hopes to be a first step toward further development and improvement of the proposed test method.

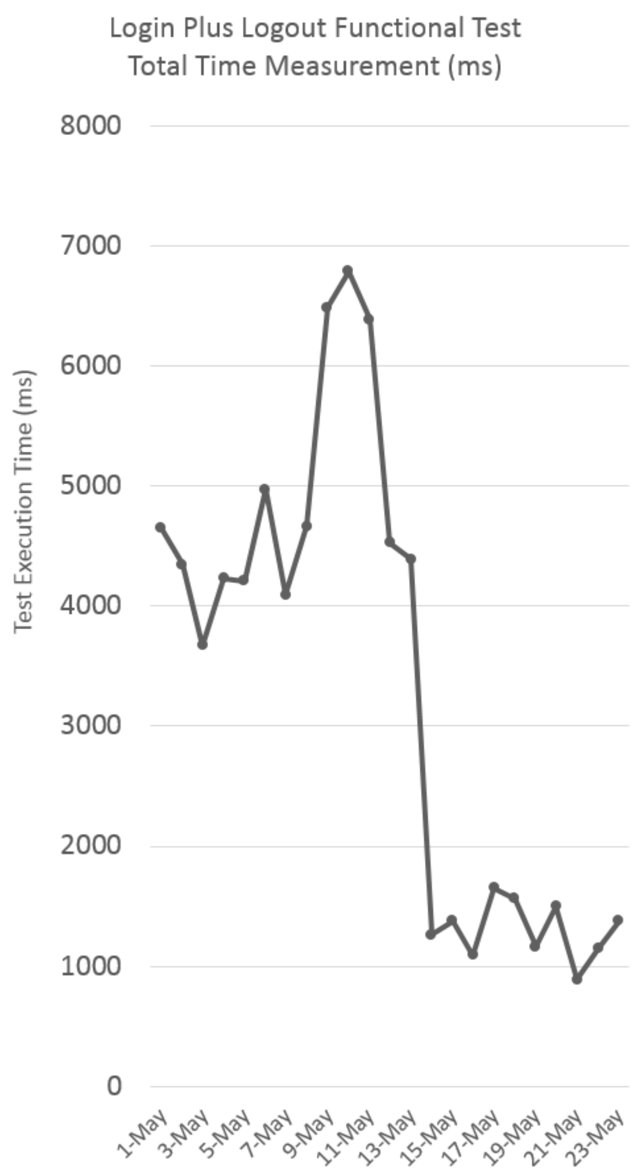

Figure 5. Execution time measurements for a simple functional test

ACKNOWLEDGEMENTS

This work is supported by the National Scientific Research Fund under the contract ДФНИ - И02/13.

REFERENCE

[1] Jones, C. Software Engineering Best Practices, 3rd ed., New York, NY, USA: The McGraw-Hill Companies, 2010.
[2] Molyneaux, I. The Art of Application Performance Testing: Help for Programmers and Quality, Sebastopol, CA, USA: O'Reilly Media, Inc., 2009.
[3] Myers, G. J. The Art of Software Testing, 2nd ed., Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004.
[4] Software & Systems Engineering Standards Committee, IEEE Standard for Software and System Test Documentation, Fredericksburg, VA, USA: IEEE Computer Society, 2008.
[5] Žilinskas A., D. Kučinskas, Implementation and Testing of an Algorithm for Global Optimizations, CompSysTech '04 Proceedings of the 5th International Conference on Computer Systems and Technologies, New York, NY, USA: ACM Inc., 2004.
[6] 15 Best Test Management Tools (2015-04-20). Available: http://www.softwaretestinghelp.com/15-best-test-management-tools-for-software-testers/
[7] Ghahrai A., Best Open Source Test Management Tools (2015-03-30). Available: http://www.testingexcellence.com/best-open-source-test-management-tools/
[8] Wikipedia, Test Management Tools (2015-04-22). Available: https://en.wikipedia.org/wiki/Test_management_tools