Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors

Per Erik Strandberg*, Daniel Sundmark†, Wasif Afzal†, Thomas J. Ostrand† and Elaine J. Weyuker†

* *Westermo Research and Development AB, Västerås, Sweden*
Email: per.strandberg@westermo.se

† *Mälardalen University, Västerås, Sweden*
Email: {firstname.lastname}@mdh.se

***Abstract*—We propose a new method of determining an effective ordering of regression test cases, and describe its implementation as an automated tool called SuiteBuilder developed by Westermo Research and Development AB. The tool generates an efficient order to run the cases in an existing test suite by using expected or observed test duration and combining priorities of multiple factors associated with test cases, including previous fault detection success, interval since last executed, and modifications to the code tested. The method and tool were developed to address problems in the traditional process of regression testing, such as lack of time to run a complete regression suite, failure to detect bugs in time, and tests that are repeatedly omitted. The tool has been integrated into the existing nightly test framework for Westermo software that runs on large-scale data communication systems. In experimental evaluation of the tool, we found significant improvement in regression testing results. The re-ordered test suites finish within the available time, the majority of fault-detecting test cases are located in the first third of the suite, no important test case is omitted, and the necessity for manual work on the suites is greatly reduced.**

I. Introduction

Software testing is performed for many reasons. Possible objectives include giving feedback, finding or preventing failures, providing confidence and measuring quality. *Regression testing* is an important part of the maintenance process for software systems that undergo periodic revision and enhancement. Whenever a system is updated, either with fixes or improvements to existing code or with new functionality, it is necessary to ensure that the system has not regressed, i.e., that the modifications have not introduced faults that might affect previously satisfactory operation of the system. Regression testing aims to detect such faults in the modified system by running all or some of the existing test cases that were used to evaluate the system's former version.

Because there is frequently not enough time, equipment or personnel available to rerun the entire test suite, regression testers focus on *selecting* the most effective subset of the test suite, and *prioritizing* or determining an efficient order to execute the selected test cases.

Testing requires resources including machine and staff-hours, hardware, and calendar time, which impacts the time to market. In our environment, software is automatically compiled, and regression testing is done nightly when the software is not being changed.

In this paper we identify problems related to this process, and describe our solutions. We found that the root cause for many, but not all, problems was insufficient time. We therefore propose a new method for prioritizing regression test cases, describe an industry-quality tool that implements it, and provide an evaluation of the method and tool as used at Westermo Research and Development AB (Westermo).

Section II gives an overview of earlier work relating to prioritization methods for regression testing. Section III describes the specific problems faced by Westermo in its original regression testing process. Section IV describes the prioritization method and the SuiteBuilder tool that implements it. The method is based on a set of individual prioritizers that each focus on a single goal, and a priority merger that combines the separate priorities. The SuiteBuilder tool facilitates the use of automated test case prioritization for nightly regression testing, and is part of the existing continuous integration framework of our software development environment.

In Section V we give detailed descriptions of the results of experimental evaluation of SuiteBuilder. In Section VI we analyze the results of the experimental evaluation, which indicate that most of the problems that motivated the creation of the prioritization method and tool were mitigated. Section VII discusses limitations of SuiteBuilder and presents possible directions for future work and extensions of the tool. Section VIII summarizes the regression testing problems, the solution arrived at, and the contributions of the paper.

II. Related work

In [21], Yoo and Harman reviewed existing research on three strategies for coping with regression: *Minimization*, to eliminate test cases from a suite if they are redundant with respect to a given set of test requirements; *Selection*, to select the subset of test cases in a test suite that are most relevant to

execute, given knowledge of changes to the software under test; and *Prioritization*, the process of reordering the test cases in a suite to favor some desirable property.

Selection can be based on properties such as code coverage [14], most likely or expected locations of faults [16], topic coverage [8] or historic test execution data [3]. Prioritization can be viewed as a type of selection, as the first tests in an ordered suite may be the only ones run. Overviews of selection and prioritization techniques appear in [21], [3], and [4].

Mathematical optimization approaches have also been proposed. Recent examples include work by Herzig et al. [9] and Mondal et al. [14], where the goal has been to optimize on cost, code coverage, test diversity and/or test duration.

Walcott et al. [20] proposes using a genetic algorithm to find the most effective suite order based on estimated fault detection ability and the expected running time. It is unknown if the approach is practically usable as the genetic algorithm was only applied to two small programs (less than 2000 LOC), with seeded faults.

Cost-effective regression testing has become particularly important due to the increased use of continuous integration processes. Elbaum et al. [2] note that traditional regression testing techniques that rely on source code instrumentation and availability of a complete test set, become too expensive in continuous integration development environments, partly because the high frequency of code changes makes code coverage metrics imprecise and obsolete. Thus recent regression test techniques are based on information sources that are more readily available and light-weight.

A recent study by Hemmati et al. [8] shows that a prioritization of test cases based on their prior fault detection capability improves test effectiveness when moving to a rapid release environment. Elbaum et al. [2] use a simple prioritization of test suites based on their prior failure in a given time window. Their results show large variance in performance across window sizes, but typically better than no-prioritization. Our approach includes a similar prioritizer, as one of our priority variables.

Saff and Ernst [18], consider several strategies for test prioritization as part of their investigation of *continuous testing*. Their goal was to reduce the non-productive time that a developer spends waiting for tests to run to completion. They compare the original test suite order and random order to orderings that give highest priority to tests that failed most recently, to tests that failed most often, and to tests with the shortest running time. They evaluated the ability of these different orderings to reduce waiting time on two moderate size code projects (5700 and 9100 LOC). For the larger project, all of the orderings performed at approximately the same level. For the smaller project, giving higher priority to tests that failed most recently resulted in a significant reduction in waiting time. A key difference between their examination of priorities and ours is that they look at each ranking method independently, while we build a test case priority by merging multiple independent rankings.

Marijan [13] described a study that used impact of detected failures, test execution time, failure frequency of a test case and functional coverage to prioritize test cases efficiently. Although the context is continuous integration, it is not clear if the proposed approach is integrated into the nightly build process. In contrast, SuiteBuilder is an integral part of the Westermo nightly test environment.

Other studies have used historical data for test prioritization. Kim and Porter [11] used a test prioritization technique based on three properties: when a test case was last executed, if a test case resulted in the identification of a fault, and the functions exercised by the test case. However, their approach was not evaluated in an industrial setting. Following this study, others have investigated history-based prioritization [3], [5], [10], [17]. However, these approaches were also neither deployed nor evaluated in industrial settings. The same is true for [5], [10], [17] which used small programs that provided useful demonstrations of proof of concept but did not provide evidence of their usefulness in practice.

While the work reported in [3] is an industrial case study with a prototype implementation of a tool, the results are still preliminary and the study was done in an offline mode. In contrast, our approach provides a fully automated tool that is integrated in the nightly build and test environment at Westermo and is being successfully used in production.

III. Problem Description

This paper addresses problems related to regression test selection and prioritization experienced at Westermo. Westermo designs and manufactures robust data communication devices for harsh environments, providing communication infrastructure for control and monitoring systems where consumer grade products are not sufficiently resilient.

A. Context

The focus of this paper is on automated testing of target devices: controlling and configuring a device running WeOS (Westermo Operating System), as an embedded system, in a real network. WeOS includes the Linux kernel, other free and open source software libraries, as well as proprietary code, resulting in an interconnected source code base containing millions of lines of code.

To run test cases on one or more *devices under test* (DUTs), a test framework has been implemented and maintained over a period of several years. The framework provides an environment for tests to be executed manually by a human tester, or for automated tests that can run and have their results recorded by the framework without human intervention. The automated testing is used in combination with several other test approaches, e.g., manual exploratory testing. This combination provides a broad and repetitive

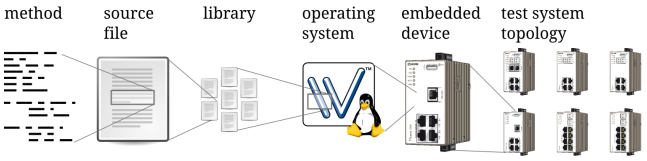

Figure 1. Layers of abstraction from source code method to topology.

regression testing on many hardware platforms by the automated testing, and also incorporates a critical perspective by experienced members of the test team, allowing them to focus on the high risk areas in each WeOS release.

In order to test different scenarios, a number of test systems with varying physical topologies have been constructed aimed at test coverage of a certain hardware product, a software feature or a customer case. The physical topologies contain from 4 to 25 devices communicating with each other using traditional Ethernet cables, optical cables, serial ports etc. Each device is built from physical components and firmwares, as well as its customized version of WeOS. The latter consists of many software libraries, which are in turn composed of source code files, each containing many methods, functions or classes. See Figure 1 for an example of a topology and an illustration of this structure.

An automated test case is implemented as a class in the Python programming language. The class includes a description of a *logical topology* that specifies requirements on the physical topology. The logical topology instructs the test framework on how many DUTs the test needs and how they are interconnected, for example, by enabling or disabling ports or by altering firewall rules. Individual DUTs are configured according to the instructions in the test cases. A test class includes methods that create setup, test execution, and tear down instructions for the test cases, and is defined with parameters that can be used to build a large number of specific test cases.

An example of a test case is our test on Power over Ethernet limits, which is a feature that lets a DUT power external devices. If the total power consumed by the external devices is above a certain limit, the DUT disables ports according to a configured priority. The test case reconfigures these priorities according to different patterns, verifies that the port with the lowest priority is disabled and that no other port is disabled.

The outcome of a test is typically pass or fail. For certain errors additional outcomes are used, for example when the test framework is unable to communicate with external hardware, when unexpected states cannot be recovered from, or if a logical topology cannot be mapped onto a physical topology. Depending on the number of WeOS versions to test, and the types of suites needed, between one and ten test suites are needed per test system per night.

B. The Problems

1) Nightly testing does not finish on time: Testing requires time and time for testing is limited. An informal study of durations of different types of testing was performed at Westermo: Manual testing of a suite of 14 test cases had been performed for a recent WeOS release. These test cases required a particularly complicated test setup with virtual machines running authentication services. Manual testing required an effort of roughly 1 hour per test case, including time for configuration, learning about the topic, reporting and interpreting the results. These test cases were subsequently automated. Running the automated cases required between 1.33 and 7.25 minutes, with an average of 2.96 minutes per test case. This is a speed up of a factor 20 when compared to manual testing. Automated test cases that can run without an embedded system, but that require communication, for example with a database, are faster. The tests for one of the modules used in SuiteBuilder required 3.5 seconds for 5 test cases, or about 0.7 seconds per test case. Unit tests are even faster. Unit testing of one of the libraries of SuiteBuilder requires 0.033 seconds for 155 test cases, or about 0.2 milliseconds per test case.

There are many reasons why testing on target devices is lengthy. For example, if a DUT needs to reboot, perhaps to assure that a configuration was stored on the disk, several seconds are required for this action alone. If the DUT has hardware such as special ports, the device may need time to load the firmware before it reaches the desired state. Tests of common ethernet protocols like Rapid Spanning Tree Protocol (RSTP) also have intrinsic time consuming characteristics due to the timing in which the protocol sends packets between devices on a network.

We gathered statistics of the durations of automated test cases in the automated test framework for three of the test systems from the first quarter of 2014 for tests that either passed or failed. The test durations ranged from 0.2 to 37 minutes, with an average of between 1.6 and 2.5 minutes. These values are presented in Table I.

Table I
DURATIONS OF TESTS IN MINUTES FROM Q1 2014 FOR NIGHTLY TESTING.

System	Fastest	Average	Slowest	Std.dev.
Test System 1	0.2	1.58	28.8	1.18
Test System 2	0.2	1.58	36.8	1.31
Test System 3	0.2	2.48	29.6	3.34

The nightly test suites for regular testing of WeOS grew over time as more test cases were implemented. Eventually, the nightly suites did not finish until after 8am, when the Westermo work day starts. When a nightly suite did not finish on time, we sometimes manually stopped it. In addition to requiring manual labor, this could also lead to

undefined states in the test systems.

2) Manual work and forgotten tests: Due to insufficient time for nightly testing, we tried to implement a workaround by manually removing tests to decrease the length of the suite. The intent was to manually add these tests into the suites during the weekend. This often led to tests that were ignored for months, sometimes leading to lost test opportunities or issues in WeOS not being detected. Moreover, any manual work with the suites was risky as one might unintentionally introduce syntax errors, or break the suite in other ways.

3) No priority for the test cases: Tests are typically named according to the functional area they aim to test. Tests were originally ordered alphabetically in the suites, so if a test had a name late in the alphabet, it was executed at the end of the test run. This default ordering had two consequences. First, tests related to a functional area late in the alphabet were canceled more frequently than tests with early names. If a late name test was not canceled but failed, the failure would consistently be noted late in the suite, providing less rapid feedback and less time for debugging. It is preferable for failures to occur early in the suite, so that feedback is faster, and more time is available for debugging.

Second, we discovered previously unknown and unexpected failures when the suites were manually reordered, similar to the observations in [12] and [22]. These failures indicated insufficient clean up in various state transitions of the test framework or sometimes also in WeOS products.

IV. The SuiteBuilder Tool

The problems described in the previous section are certainly not unique to regression testing at Westermo. Others have noted them and looked for solutions, as surveyed in Section II. Our goal was to solve all of the above-mentioned problems while maintaining Westermo's existing nightly testing framework and adhering to the following principles: (i) the solution should be easy to extend, (ii) if the solution included weighted priorities, then the weights must be easy to change, (iii) it must be possible to explain the approach to knowledgeable testers in less than 15 minutes.

These principles reflect the need for the system to be flexible enough to allow for easy modification as priorities change.

Our solution was a method that includes a vector of priorities, each of which can be assigned to a test case independently. The method merges the priorities of each test case into a single final value which is used for the final ordering of the entire regression test suite. Some priorities are assigned statically by humans, others are derived automatically from historic test outcomes or source code changes.

The priority merging process is controlled with weights for each priority that can be adjusted by the test team, according to which goals are deemed most important at a particular time.

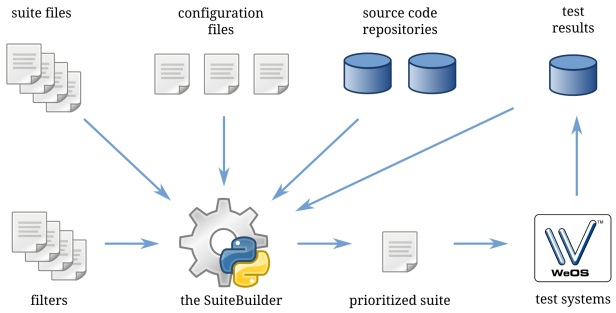

Figure 2. Overview of the workflow of the SuiteBuilder tool.

A. Approach and Overall Workflow

The fully automated workflow of SuiteBuilder (also illustrated in Figure 2) can be summarized in five main steps: (i) *Collection of raw test suites.* Upon invocation, SuiteBuilder parses the files that describe available tests, and builds a suite that includes all tests currently in use. (ii) *Assigning priorities.* Next, a set of prioritizers assign a set of priorities to each test case in the raw test suite. This process will be explained in detail in Section IV-B1. (iii) *Merging priorities.* Once all test cases have been assigned priorities, each test case is given a final priority by merging the individual priorities. The outcome of this step is a list of all test cases in the raw test suite, sorted based on their assigned final priority. The priority merging process is described in Section IV-B2. (iv) *Selecting the final suite.* Given the prioritized list of test cases, and the time allotted for testing, the final suite is selected through a two step process. First, the list of test cases is filtered based on the capabilities of the test system in question. Second, the remaining test cases are placed into the suite in order of priority, as long as the tests already placed in the suite have not exhausted the allotted testing time. The time estimated for a partially-built suite is the sum of the individual test case times, based on their prior running times. If there is no prior history for a test case, a default of 3 minutes is used. More details of the selection of the final suite are provided in Section IV-B3, (v) *Executing the suite.* The final suite is executed by the test framework, and the results from the test session are stored in the test results database.

The above steps are repeated for each release to be tested, for each test system, and for each accompanying test suite.

B. Prioritization, Priority Merging and Suite Selection

The following sections explain the prioritization of test cases using prioritizers, how different priorities are merged and combined into a final priority for each test case, and how the list of prioritized test cases are combined into a final suite.

1) Assigning Priorities: TestPrioritizer assigns a static prioritization for an individual test, determined by experienced WeOS developers, written in configuration files that are rarely modified. It assigns a low baseline value for most tests, an increased priority for tests that can only run on limited test systems, and a decreased value for tests that can be run on most or all test systems.

The *TagPrioritizer* assigns a priority to all members of a group of tests. Tests are organized into groups by WeOS developers, based on domain and test framework knowledge using a tag concept. A detailed example of this concept is given in Section IV-C2. Like the TestPrioritizer, this is a static prioritization made with configuration files that are rarely modified. The TagPrioritizer gives a rough prioritization for an entire group of tests, while the TestPrioritizer provides a fine-tuned priority that works on individual tests.

Failing tests might indicate issues in WeOS, so a test that has failed recently and/or frequently will have its priority increased with the *FailPrioritizer*. By using threshold values, it increases the priority on test cases with more than a specific number of failures within a specific number of days, Section IV-C3 explains this further.

The *RecentPrioritizer* increases priority for tests that have not been selected recently. A goal of SuiteBuilder is to assure test circulation so that no tests consistently have low priority and hence never get tested.

The software under test control physical machines and generally do not permit the measurement of code coverage while the software runs on these target devices. This type of monitoring would impact the program execution in ways that would make the value of the test results very low [19]. However, the source code repositories of WeOS and the test framework provide very detailed logs describing where code changes are located in the software under test, and also in the test framework. Two *SourcePrioritizers* parse these logs for tags to increase the priority of tests associated with code areas that have recent changes. Section IV-C3 contains an example of such a log message.

2) Merging Priorities: The purpose of the *PriorityMerger* is to derive a single final priority for each test case, based on the values provided by the individual prioritizers. SuiteBuilder uses a weighted average of the prioritizer values. In case a test lacks a value from a particular prioritizer, that prioritizer is not included in the weighted average for that test. This is similar to what was proposed by [3], [5], [10].

The individual prioritizers use constants with numerical values for their priorities: `LOWEST`, `LOW`, `MEDIUM`, `HIGH` or `HIGHEST`, with numerical values 1, 3, 5, 7 and 10.

Table II shows an example of merging three priorities for four test cases. Prioritizer A provides values for all four test cases and assigns priorities with the numerical values 5, 3, 7 and 5 respectively. Prioritizer B only assigns priorities to two of the test cases and Prioritizer C to three of them.

The final priority for each test case is the weighted average of its individual priorities. T1, for example, is assigned a priority by only one prioritizer, so the value from this prioritizer is used. The test case T2 gets the final priority $\frac{9 \cdot 3 + 2 \cdot 1}{9+2} = 2.6$.

Table II
EXAMPLE TO ILLUSTRATE HOW THE PRIORITYMERGER WORKS.

	T1	T2	T3	T4
Prioritizer A (weight 9)	5.0	3.0	7.0	5.0
Prioritizer B (weight 5)	-	-	5.0	7.0
Prioritizer C (weight 2)	-	1.0	5.0	7.0
Final Priority	5.0	2.6	6.1	5.9

In order to get initial weights, we created scenarios where the prioritizers competed over tests. For example "if a test is not tested recently *and* has code changes, then it is more important than a failing test". This was a good start, but led to frequent frustrations when expected tests were not included in nightly testing. This led to many initial changes in the weights of the prioritizers, until we reached the consensus that, for us, the most important tests are the ones that recently failed. Now that the weights have stabilized, the FailPrioritizer is the prioritizer with the most impact. An overview of the weights is presented in Table III.

Table III
RELATIVE WEIGHTS OF THE PRIORITIZERS.

Prioritizer	Weight
FailPrioritizer	9.3
SourcePrioritizer (WeOS)	6.1
SourcePrioritizer (Test Framework)	6.1
RecentPrioritizer	4.1
TestPrioritizer	2.6
TagPrioritizer	2.2

3) Selecting the Final Suite: SuiteBuilder uses four main parameters to select which test cases to include in the final suite: (i) the raw test suite with test cases ordered by final priority, (ii) knowledge of which test cases can be run on which test systems, (iii) the expected execution time of each test case, and (iv) the desired duration of the suite in question.

The list of potential test cases is first filtered based on the capabilities of the test system in question. Filters for experimental tests and experimental functional areas, as well as tests with known issues limit the number of test cases to be considered. We have learned from experience that the latter two types of tests sometimes cause the test framework to enter undefined states from which it cannot recover. If these tests are run first, this sometimes causes the remaining test results to be lost. Therefore, suites containing these types of tests are run in isolation, after the regular nightly testing.

Execution times of the test cases are estimated based on

prior execution times. When historic data on test duration is not available, for example for new tests, a default duration of 3 minutes is used, as this is slightly higher than the average of all test execution times. This avoids the risk of adding too many test cases so that a suite generally will finish on time.

The desired duration of the suite also needs to be provided. At Westermo, the desired duration of the suite in question is typically a portion of the full night. Examples of expected and actual durations are presented in Figure 3. The duration of a test does not affect the prioritization, but if slow tests have a high priority then the final suite will contain fewer tests as the total duration of the suite was determined a priori. Test cases are placed into the final suite with a greedy algorithm in priority order, until the allocated duration is reached. When more than one test case has the same priority, alphabetical sorting is used as a tie breaker.

C. SuiteBuilder Tool Implementation

The entire SuiteBuilder solution is implemented in Python as 15 modules with a total of a few thousand lines of code. This section provides details of the tool, an accompanying test result database and test suite library.

1) Test Result Database: Originally, the test framework reported results from nightly testing in conventional log and report files. In 2012, we started placing results into a relational database with generous reading rights, to make it easily accessible by staff and possible future tools. After about 20 months, we had accumulated more than one million test outcomes.

The Test Result Database allows us to rapidly answer questions like: "When was a particular test case last successfully run on a specific test system?", "How many times has a particular test case failed on a specific test system in the last few days?", "What is the long term pass/fail ratio for a particular product version over time for a particular set of functional areas?", or "What is the average duration for a particular test case?" The result database provides one of the major sources of data for the prioritizers.

2) Tags: A central concept of SuiteBuilder is the use of tags to group tests together in a form of key-value pairs, where the key is a tag and the value is the set of tests associated with this tag. A hypothetical example could be the tag *backup* with the associated tests backup1, backup2 and backup3. These *tag to tests mappings* are used by three of the prioritizers (TagPrioritizer and both SourcePrioritizers), two of which (both SourcePrioritizers) use them together with source code history to adjust the priority of the tagged tests. The set of tags was manually created by inspecting internal documentation, names of source code files, and by interviewing developers. A tag can have an assigned priority, which is inherited by all its associated tests.

A test can be associated with several tags, so that its priority can be governed by any one of the tags. The tags are manually configured in a configuration file that contains entries with a tag and a number of patterns to match test paths:

```
- tag: poe
  tests:
    - src/test/poe/*
    - src/test/*/*poe*
```

3) Prioritizers: All prioritizers share the *BasePrioritizer* class as a common base class. It handles information on test cases, and can be asked what the priority of a test case is. The idea is to allow a prioritizer to *increase* the priority of a test case if the priority is elevated from the perspective of this prioritizer, for instance if the test recently failed. The prioritizer may also *decrease* the priority if the test should be suppressed, for example, if it has frequently been part of nightly testing. It is also possible for a prioritizer to *remain silent* if the test is unknown from the perspective of this prioritizer.

This example illustrates how the base class can be extended into a simple prioritizer with only four Python statements:

```
from priority import *

class MyLowPrioritizer(BasePrioritizer):
    def get_prio(self, path, params):
        return PRIO_LOW
```

The prioritizers with a static priority (TestPrioritizer and TagPrioritizer) parse the required configuration files in order to give their priorities. FailPrioritizer and RecentPrioritizer instead query the Test Results Database for theirs. The FailPrioritizer sets the test case priorities based on when the test last failed. A test that failed within the past 3 days is assigned a high value. Failures that occurred longer ago are assigned correspondingly lower priorities, as shown in Table IV. Test cases with few failures are given a priority lower than the default in order to avoid spending time on test cases that seem to consistently pass.

Table IV
PRIORITIES SET BY THE FAILPRIORITIZER.

Last fail	Priority
1-3 days ago	avg(highest, high)
4-5 days ago	high
6-7 days ago	avg(high, medium)
8-10 days ago	medium
11-13 days ago	avg(medium, low)
14-16 days ago	low
17-20 days ago	avg(low, lowest)
>20 days ago	lowest

The RecentPrioritizer protects against two extreme situations: test cases that are in danger of never being executed as their priority is pushed lower and lower, and test cases that have been executed repeatedly without failing in recent days. This prioritizer searches for test results and assigns

a priority based on how many days in the past a test last passed. The lowest priority on a given test system is assigned to test cases that have been selected three nights in a row on this system.

The two SourcePrioritizers (one for changes in WeOS code and a second for test framework changes) parse the news feed from the source code repositories, extract details on the names of files altered, and also parse the messages of the change sets. A typical content of such a feed entry is illustrated below. The entry is treated as plain-text, and the search is a plain-text scan for an occurrence of the tag. Because this entry contains the word **poe**, it will affect any tests that are in the group with tag "poe".

```
Date: October 11, 2016
Author: Per Erik Strandberg
Message: Fixed a bug in the POE wrapper.

Changed files:
 + src/lib/POE.py
```

This illustrates how the tag key is also used as a trigger word. In this case, all tests associated with the tag poe can expect a priority increase since we have detected changes relevant to the poe tag. Like the Fail and RecentPrioritizers the SourcePrioritizers set their priority based on how far back in time they need to search. The highest priority is given to a tag that has received changes within one day, as shown in Table V.

Table V
PRIORITIES SET BY THE SOURCEPRIORITIZERS.

Last code change	Priority
1 day ago	avg(highest, high)
2-3 days ago	high
4-7 days ago	avg(high, medium)
8-14 days ago	medium
>14 days ago	(no priority assigned)

4) Suite Handler Library: Partial suites and test system capabilities are input data to SuiteBuilder. These specify what test cases each test system can run and may limit the number of possible tests for each test system, based on hardware requirements. This processing of configuration files, and export to suite file is implemented in a *Suite Handler Library*.

V. EXPERIMENTAL EVALUATION

Our experimental evaluation of SuiteBuilder examines the extent to which the proposed method implemented by the tool solves Westermo's original regression testing problems.

A. Nightly testing does not finish on time

The first, and most problematic, issue addressed by SuiteBuilder concerns the selection of test cases given that a limited amount of time was allocated for nightly regression testing. To address this problem, SuiteBuilder selects test cases based on the prioritization algorithm. The expected running time for an individual test case is based on the times of previous runs of the case. A potential test suite is allocated a certain amount of running time, and test cases are added to the suite in order of priority as long as there is at least some time left. When the suite is completed, its expected duration is the sum of the expected times of all its test cases. Since the expected time of the last test case added can be longer than the suite's remaining time, the final expected duration of the suite is typically longer than its allocated duration.

In order to determine if the suites created by SuiteBuilder finish on time, we compared the actual running time durations of 701 test suites created by the tool during one of the months after SuiteBuilder began nightly usage to the expected durations of those suites.

Figure 3 shows the expected and actual durations of the 701 suites. The expected durations are plotted as a blue line, and the actual durations as red dots. The running time of the great majority of suites was very close to their expected time. For some suites, the actual time is significantly lower than the expected time (red dots that are far below the blue curve). These are typically suites for which no test cases could be found (such as an experimental suite when there are no experimental tests), suites that crashed, were aborted or suites that had poor estimates, perhaps because they had one of the above issues in the past.

We found that no suite took more than 120% of its expected duration. The largest underestimation in absolute time between an expected and an actual time was 50 minutes. Of the 701 investigated suites, 12 required between 105% and 120% of the expected duration, 433 required between 95% and 105% of the expected duration, 211 required less than 95% of the expected duration, and 45 had no duration at all. Suites in the last category were either empty or aborted at the start.

This evaluation has given us positive feedback that most test suites are now finishing close to their expected time and therefore typically run to completion without having to be aborted at start of business the following day.

B. Manual work and omitted tests

The second problem addressed by SuiteBuilder concerned the need for substantial manual effort for preparing suites for nightly testing, and the risk of removing and eventually totally omitting certain tests in this process.

In order to investigate whether tests had been omitted from the test suites by SuiteBuilder, we first collected all the tests that had been part of the regular nightly testing for nine months preceding and three months following the introduction of SuiteBuilder. We then compared this 12-month set of tests against all the tests generated by SuiteBuilder in the second month after its introduction.

By comparing the two sets of test cases, we can assess to what extent tests are omitted by SuiteBuilder, as they often

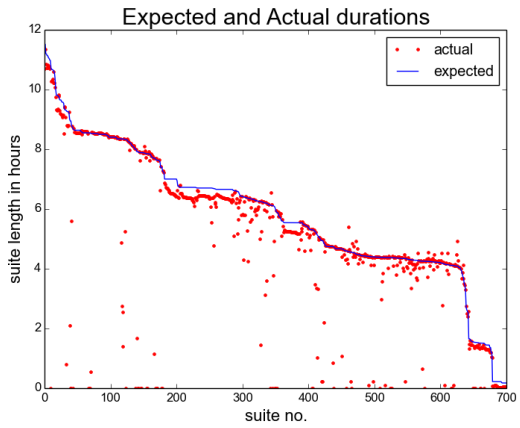

Figure 3. Actual and expected durations of suites.

were when the manual process was used. In particular, if many of the test cases in the 12-month set are missing from the tests generated by SuiteBuilder in a single month, this would indicate that SuiteBuilder's selection process has not solved the omitted test case problem. Conversely, if none or very few test cases from the 12-month set are missing in the SuiteBuilder set, this would indicate that SuiteBuilder does not tend to systematically omit test cases over a long period of time.

On initial examination, it appeared that almost 100 test cases that were present in the 12-month set had been omitted from the SuiteBuilder set. However, after careful examination we discovered that in fact no test case had really been systematically omitted by SuiteBuilder. The missing test cases were explained as follows.

Forty-one test cases in the 12-month set that at first seemed to be missing from the SuiteBuilder set turned out to be actually identical to tests in the SuiteBuilder set, except for inconsequential changes such as parameter order. Nine test cases in the 12-month set had either been deliberately renamed or removed. Eight tests were blocked due to changes in WeOS. These tests had been modified and reintroduced in the nightly suites after the investigated period ended. Six test cases, or the features in WeOS they covered, had become unstable. These tests had been moved to the experimental suites for investigation.

Two test cases belonged to test scripts where unique identifiers had accidentally been duplicated so that they were not included in database queries. One test case was blocked due to physical wear and tear; it rewrites a memory area on a physical disk device that can only withstand a few hundred rewrites, and the test should not have been included in the nightly testing. Another test case had been blocked due to special topology needs. It was later reintroduced. One test case was missing due to manual labor *prior to* the implementation of SuiteBuilder. It had been moved around in the suite to investigate if the suite order was significant for an infrequent and sporadic error to occur. In this manual shuffling, it had accidentally been removed and later forgotten.

Finally, because the period covered by the 12-month set extended beyond the one month period of the SuiteBuilder set, 28 new test cases that were created during that additional time could not have been included in a regression suite by SuiteBuilder because they didn't exist yet.

Table VI summarizes these omitted tests.

Table VI
TESTS EXCLUDED BY SUITEBUILDER

Reason for exclusion	Number of excluded tests
SuiteBuilder tests identical up to parameter order	41
Tests were renamed or removed from regular testing	9
Blocked tests	8
Unstable features; tests moved to experimental suite	6
Non-executable tests due to duplicate identifiers	2
Test blocked due to potential physical equipment wear	1
Test blocked due to topology requirement	1
Test case missing because of incorrect manual removal	1
Tests introduced after Suitebuilder application period	28
TOTAL of excluded cases	97

The conclusion from this experiment is that during the single month, SuiteBuilder selected all tests it should have from the preceding nine months, and did not allow any test to be systematically omitted for months at a time.

In order to determine if the need for manual work had decreased, we relied on expert opinion by interviewing test specialists who have spent a lot of time manually working with the suites. Some relevant answers were: (i) "There is still manual work, for example when introducing a new test or when moving tests between suite types." (ii) "Before the implementation of SuiteBuilder, suites were manually altered to allow more testing during the weekends. This was not always done. Tests were easier to forget before the implementation of SuiteBuilder. Another improvement is that the length of the suites are changed dynamically depending on how much time we have available." (iii) "An inconvenience that existed before SuiteBuilder was when developers or testers came in to the office in the morning, looked at partially complete test suites and wanted the rest of the tests aborted, so that debugging could start on a particular test system. Suites finish on time now and there is no need to perform these steps." (iv) "Yet another issue was when developers came to work in the morning and tests they were particularly interested in were still in queue."

The feedback given by the test specialists show an obvious reduction in the risky manual work, which is a positive result for Westermo.

C. No test case priority

The third and final problem addressed by SuiteBuilder is the lack of a meaningful prioritization of test cases within a test suite. Consequently, one of the SuiteBuilder goals is to have *different* priorities associated with test cases in order to reflect their relative importance.

We first examine how rankings set by all prioritizers in one suite on one test system for one night are merged to a single test case priority, thus yielding a prioritized test suite. Second, we evaluate the meaningfulness of the prioritization by investigating how the final priority correlates with fault detection.

The plot in Figure 4 illustrates the actual priorities on test cases in the one night/one test system test suite mentioned above. Tests in the suite now have a concept of priority, and the test cases with the highest priorities are the ones we determined to be most relevant in our environment.

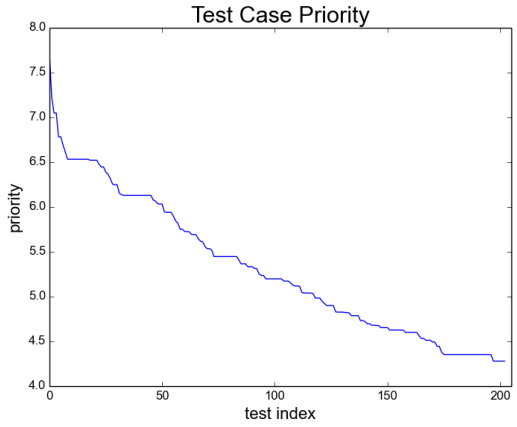

Figure 4. Priority of test cases of a test suite of 203 test cases.

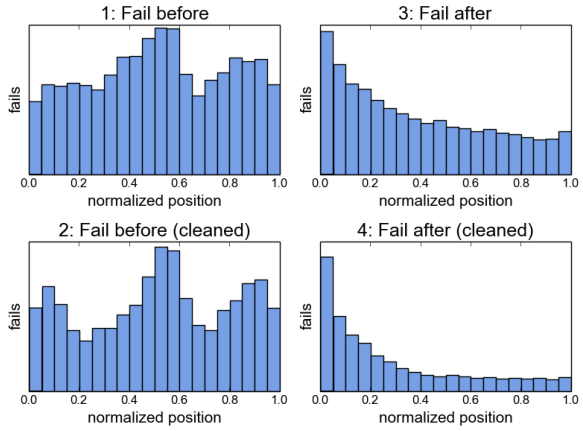

Figure 5. Normalized fail distributions before and after the introduction of SuiteBuilder. The top row includes all test suites. In the bottom row, short and broken suites have been removed from the data set.

In order to determine if SuiteBuilder had altered the distribution of the failing tests, we gathered test results from nearly four years of regular nightly testing, excluding any experimental test suites. We compared the distribution of failing tests in the 6758 suites that were run during the 23 months before the introduction of SuiteBuilder against the distribution of failures in the 8930 suites run during the 23 months once we began using SuiteBuilder.

Our goal is to measure the percent of all failures that are detected by tests relative to their positions in a suite. The suites are typically not of the same length, so in order to get the failure distribution of all suites, we scaled the index of each failing test to a number between 0 and 1, so that the normalized position can be expressed as $p_{norm} = \frac{p}{len}$, where p represents the test's position in the suite and len the number of tests in the suite.

Figure 5 shows the failure distributions of the pre- and post-SuiteBuilder test cases. Each bar represents the failures detected by 1/20 of all the test cases. The first bar is the failures that are detected by the first 5% of test cases that run in all suites over the relevant 23 month period.

Imagine that the execution period of each suite is divided into 20 segments with equal numbers of test cases in each segment (the segments don't have to use equal time), and consider the bars to be numbered from 1 to 20. Bar i represents all of the tests from all suites that ran in segment i of their suite. The height of the bar represents the percent out of all failing tests over the entire 23 month period that occurred in segment i.

These distributions are illustrated in plots 1 and 3 in Figure 5. Before SuiteBuilder, the first third of the tests in the suites contained fewer than a third of the failures. Once SuiteBuilder was used, the first third contained half of the failures.

Sometimes the nightly testing was prematurely aborted. This could happen when WeOS was left in an unstable state before nightly testing. This also might occur when the test framework is in an unstable state, or that the test framework had not compensated for recent changes in WeOS. The effect is typically a "broken suite", that is either aborted prematurely, or cause all or many of the test cases in the suite to fail. Since short suites with many failures would result in failures all over the distribution once the position is normalized, we suspected that these initial distributions (plots 1 and 3 in Figure 5) might hide more meaningful distributions.

Therefore, in order to determine the performance of SuiteBuilder in terms of failing test distribution, we removed "broken suites" from the data set used for this assessment. This was done in a two step process. We first eliminated suites containing fewer than 20 tests, and second, we removed any of the remaining suites with a higher than 40% failure rate. This "cleaning" process removed a sizable percent of the failures so that the cleaned suites contained

54% of the original failures in the pre-SuiteBuilder period, and 39% of the original failures in the post-SuiteBuilder period.

The failure distribution of the cleaned suites before SuiteBuilder was introduced remained poor, with only about 27% of the failures in the first third of the suites. However, after the introduction of SuiteBuilder, the cleaned suites identify just above two thirds of the failures in the first third of the suites. The cleaned distributions are illustrated in plots 2 and 4 in the bottom row of Figure 5.

Our conclusion is that SuiteBuilder has clearly improved the failure distribution, leading to much earlier identification of failures in a typical suite and therefore these failing test cases are far more likely to be included in the nightly regression test suite.

VI. Discussion

As shown by the evaluation in the previous section, SuiteBuilder has effectively addressed most of the problems related to lack of time in nightly regression testing at Westermo. The evaluation also shows the feasibility of an automated nightly regression test selection process in an industrial environment.

The key aspects and benefits of the SuiteBuilder project are: (i) *System Level Regression Testing*: The Westermo test framework runs testing on target device topologies. Tests are run nightly to test for regressions. (ii) *Automated Regression Test Selection*: SuiteBuilder is implemented as a system of a few Python modules. It rapidly builds suites for many test systems by using requirements for suites, a set of test priorities, and properties of hundreds of existing automated tests. The time required to build the suites is small compared to the time needed to run the test cases. (iii) *Automated Testing*: The outputs, and some of the inputs, of the SuiteBuilder are suite files that are used by the test framework without human intervention. (iv) *Industrial Context*: SuiteBuilder solves real problems in an eco-system of build servers, test systems with topologies of target devices, source code management systems, a test result database, and human beings.

SuiteBuilder was placed in production at Westermo in the late spring of 2014. Since then it has been used continuously to build suites for nightly testing.

A number of significant changes have been made as we gained experience: (i) We reconfigured the PriorityMerger to increase the impact of the FailPrioritizer. (ii) Indices in the database tables and other optimizations in the database queries were introduced to speed up queries. (iii) The SourcePrioritizer for the test framework was introduced.

The majority of the issues with SuiteBuilder have been of the following types: (i) Unbalanced priority merging: "Test X failed on test system Y last night - why was it not tested tonight?". This has been mitigated by altering the balance of the prioritizers. Or (ii) Syntax Errors: If a test is added in a suite file it is now more complicated to realize if there are issues. This is because the suites are now built by an automated system some time in the evening and not manually as soon as tests are added. This has been mitigated with thorough self tests of SuiteBuilder, as well as gated commits that rejects code changes with syntax errors.

The building of suites is not time critical, as it can be done some time before the nightly testing is to start. The time needed to build a typical suite is less than 15 seconds, and factors such as database caching can reduce this further. This duration is low (seconds) in comparison to the length of the suites (hours).

The architecture of the SuiteBuilder allows for reduction, but also extensions. We believe that similar approaches, perhaps as simple as a system of one prioritizer, may be useful for many organizations. We would like to see more studies on topics like: "How to best merge priorities for other types of suites?", "What other prioritizers could be useful?", "Are suite characteristics altered with different PriorityMergers?", "How does one best incorporate minimization?", and "Is this framework general enough to support plug-and-play of prioritizers and flow-modules such as minimizers?"

VII. Limitations and Future Research

While the method detailed in this paper has shown positive effects, and its evaluation can be seen as a first indication of feasibility, there are several extensions that could be considered to further improve the performance of SuiteBuilder.

A. Other Possible Suite types

Internal discussions on possible usages of SuiteBuilder has proposed that we could modify the configuration to allow other types of suites, for example: *Smoke Test Suite*: A short suite for the continuous integration framework to be executed after a source code commit. Perhaps the approach in [1] and the SourcePrioritizers are of particular interest? *Weekend Test Suite*: Perhaps tests that are particularly slow should be included *only* in the weekend suites. *Release Test Suite*: Perhaps certain test cases should only be included once per release cycle?

B. Other Possible Prioritizers

We speculate that other prioritizers could be of interest. A *LockPrioritizer* could include certain tests in every suite. A *ReleasePrioritizer* could focus on the new features in a code branch.

Discussions of *where* in the source code to invest the test effort are presented in for example [15], [16], where the authors make significant use of an issue tracker. The SuiteBuilder tool is trying to solve a related problem without connection to an issue tracker, instead we use a tag heuristic and SourcePrioritizers to focus test effort on certain parts of the test suite. However, given an explicit issue tracker connection, one could conceive of an *IssuePrioritizer* that

directly targets bug fixes. If an issue is resolved, or later reopened, we want the associated tests to get an increased priority.

In [3] the authors use what could correspond to an *AgePrioritizer*. It is used to decrease the priority of old test cases. This is an interesting concept, also supported by [6], that seems to overlap with the SourcePrioritizers and the FailPrioritizer. A test that rarely fails and that has not had its source code altered for a long time could be considered old, and have its priority decreased over time.

C. Minimization and Diversification

The SuiteBuilder does not include minimization, we believe that a *SuiteMinimizer* could easily be added. We investigated the number of outcomes from test *classes*, as opposed to test cases, and found that some were over-tested. The most over-tested class outnumbers most others by a factor of 10, some by a factor of 100.

Over-testing is perfectly normal in many scenarios as we want high-priority tests run more often. In our case the reason was the ability of the test class to handle many parameter combinations, and that it could be executed on most test systems.

In addition, preliminary evidence suggests that there might be some value in altering the order of the tests in the suites. We discovered this when suites were no longer testing from A to Z. The authors of [14] and [7] suggest that the concept of *diversification* is a promising approach. We speculate that diversification somehow overlaps with minimization and that if one implements one concept, then you also get the other.

D. Other Approaches for Priority Merging

Using a weighted average in the PriorityMerger is an intuitive and flexible way of merging the priorities. What advantages and disadvantages come with this approach compared to a mathematical optimization approach such as the one investigated in [9], or approaches including a cost-benefit ratio analysis such as the one investigated in [14]?

VIII. Conclusions

In this paper we describe an implementation and evaluation of the SuiteBuilder tool that aims at solving problems we encountered at Westermo with nightly regression testing. These problems included: (i) *Nightly testing did not finish on time*: SuiteBuilder addressed this problem adequately so that time allotted for the nightly run, typically closely matched the time needed to run the generated regression suite. This is shown in Figure 3. Consequently, nightly testing generally finishes prior to the start of business, and we no longer need to manually interrupt testing. (ii) *Manual work and omitted tests*: We have seen that the test circulation is now adequate. Test cases no longer go for extended periods of time without being included in the test suite. As a result, most of the risky manual work has been removed. There are, however, a few manual tasks that remain such as the introduction of new test cases. We emphasize here that while SuiteBuilder automatically generates regression test suites, it does this by selecting from among existing test cases. It does not generate new test cases. This must be done manually, (iii) *No priority for the test cases*: We now have a priority distribution, as illustrated in Figure 4. The output from SuiteBuilder is one or more suite files with tests that are sorted in priority order based on the criteria our team determined were most relevant to our needs. The majority of the failing tests are now located in the first third of the suites, as illustrated in Figure 5.

These problems were solved by implementing SuiteBuilder, a framework of prioritizers that provide priorities based on different characteristics of the test cases. The individual priorities are merged into one overall prioritized suite where tests are selected in order until the allocated time has been consumed. In Figure 2 we illustrate the overall flow through this process of nightly testing. Steps could be added or removed depending on the needs of the organization. We believe that SuiteBuilder is an extendable framework that could be used by many organizations in many different scenarios.

Acknowledgments

The authors thank colleagues at Westermo R&D, for their cooperation and assistance. Thanks in particular to Johan Beijnoff for eliciting requirements, discussing the design and evaluating the outcome of SuiteBuilder; and also to Raimo Gester and Peter Johansson for encouragement.

This work was in part supported by the Swedish Research Council through grant 621-2014-4925 and the Swedish Knowledge Foundation through grants 20130258 and 20130085.

References

[1] E. Dunn Ekelund and E. Engström, “Efficient regression testing based on test history: An industrial evaluation” in *Proceedings of the 2015 International Conference on Software Maintenance and Evolution (ICSME’15)*, 2015.

[2] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments” in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE’14)*, 2014.

[3] E. Engström, P. Runeson and A. Ljung, “Improving regression testing transparency and efficiency with history-based prioritization–An industrial case study” in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST’11)*, 2011.

[4] E. Engström, P. Runeson and M. Skoglund, “A systematic review on regression test selection techniques” in *Information and Software Technology* 52.1 (2010): 14–30.

[5] Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi and S. Parsa, "Prioritizing test cases for resource constraint environments using historical test case performance data" in *Proceedings of the 2nd International Conference on Computer Science and Information Technology (ICCSIT'09)*, 2009.

[6] R. Feldt, "Do system test cases grow old?" in *Proceedings of the 7th International Conference on Software Testing, Verification, and Validation (ICST'14)*, 2014.

[7] H. Hemmati, A. Arcuri and L. Briand, "Achieving scalable model-based testing through test case diversity" in *ACM Transactions on Software Engineering and Methodology* 22.1 (2013): 6:1–6:42.

[8] H. Hemmati, Z. Fang and M. V. Mäntylä, "Prioritizing manual test cases in traditional and rapid release environments" in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST'15)*, 2015.

[9] K. Herzig, M. Greiler, J. Czerwonka and B. Murphy, "The art of testing less without sacrificing quality" in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015.

[10] A. Khalilian, M. Abdollahi Azgomi and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data" in *Science of Computer Programming* 78.1 (2012): 93–116.

[11] J-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments" in *Proceedings of the 34th International Conference on Software Engineering (ICSE'02)*, 2002.

[12] Q. Luo, F. Hariri, L. Eloussi and D. Marinov, "An empirical analysis of flaky tests" in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE'14)*, 2014.

[13] D. Marijan, "Multi-perspective regression test prioritization for time-constrained environments" in *Proceedings of the 2015 International Conference on Software Quality, Reliability and Security (QRS'15)*, 2015.

[14] D. Mondal, H. Hemmati and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection" in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST'15)*, 2015.

[15] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system" in *ACM SIGSOFT Software Engineering Notes* 27.4. (2002): 55–64.

[16] T. J. Ostrand, E. J. Weyuker and R. M. Bell, "Predicting the location and number of faults in large software systems" in *IEEE Transactions on Software Engineering*, 31.4 (2005): 340–355.

[17] H. Park, H. Ryu and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing" in *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement (SSIRI'08)*, 2008.

[18] D. Saff and M.D. Ernst, "Reducing wasted development time via continuous testing" in *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE'03)*, 2003.

[19] M. Tikir and J. Hollingsworth, "Efficient Instrumentation for Code Coverage Testing", in *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA'02)*, 2002.

[20] K. Walcott, M. Soffa, G. Kapfhammer and R. Roos, "Time-Aware Test Suite Prioritization", in *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA'16)*, 2006.

[21] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey" in *Software Testing, Verification and Reliability* 22.2 (2012): 67–120.

[22] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption" in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.