Automated System Level Regression Test Prioritization in a Nutshell

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

Abstract

Westermo Research and Development AB has developed an automated tool called SuiteBuilder to determine an effective ordering of regression test cases. The ordering is based on multiple factors such as fault detection success, interval since last execution and code modifications. SuiteBuilder has enabled Westermo to overcome numerous problems in regression testing, including lack of time to run a complete regression suite, failure to detect bugs in a timely manner, and tests that were repeatedly omitted. This paper describes the tool and the following successful results achieved during the first two years of tool use: re-ordered test suites finish within the available time, the majority of fault-detecting test cases are located in the first third of suites, no important test case is omitted, and the necessity for manual work on the suites is greatly reduced.

1. Introduction

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

Westermo designs and manufactures robust data communication devices for harsh environments, providing communication infrastructure for control and monitoring systems for which consumer grade products are not sufficiently resilient. In Westermo's development environment, software is automatically compiled, and regression testing is done nightly. The effectiveness of the regression testing is enhanced by a Westermo tool called SuiteBuilder[1] that prioritizes the set of existing test cases. The tool automates and improves test case selection for nightly regression testing as part of the existing continuous integration framework of Westermo's software development environment. SuiteBuilder consists of a set of *prioritizers* and a *priority merger*. Each individual prioritizer focuses on a single goal, and the merger weights and combines the individual priorities. Because of the successful use of SuiteBuilder in the operational Westermo software development and testing environment during 2014 to 2016, the tool can be classified to be at Technology Readiness Level 7 (TRL 7).[9]

2. Regression Testing Challenges

Cost-effective regression testing has become particularly important due to the increased use of continuous integration processes. Traditional regression testing techniques rely on source code instrumentation and availability of a complete test set. These techniques are too expensive in continuous integration development environments, partly because the high frequency of code changes makes code coverage metrics imprecise and obsolete.[2] Thus modern regression test techniques are based on information sources that are more readily available and light-weight, such as using a simple prioritization of test suites based on their prior failure in a given time window.[3] Yoo and Harman[4] reviewed existing research on the three main strategies for building effective regression suites:

- *Minimization*: eliminate test cases if they are redundant with respect to a given set of test requirements;
- *Selection*: select the subset of test cases that are most relevant to execute, given knowledge of changes to the software
- *Prioritization*: reorder test cases to favor some desirable property.

Westermo's robust data communication devices run WeOS, an operating system including the Linux kernel, other free and open source software libraries, and proprietary code, resulting in a multi-million line source code base. A number of WeOS versions have been developed and maintained.

To run test cases on one or more *devices under test* (DUTs), Westermo has built a test framework that provides an environment for tests that a human tester can run manually, or for automated tests that the framework can run and record results without human intervention. In order to test different scenarios, a number of test systems with varying physical layouts have been constructed to test each hardware product, software feature or customer case. The physical layout contains from 4 to 25 devices which communicate using Ethernet cables, optical cables, serial ports, etc. Each device is built from physical components and firmware, as well as its customized version of WeOS (see Figure 1).

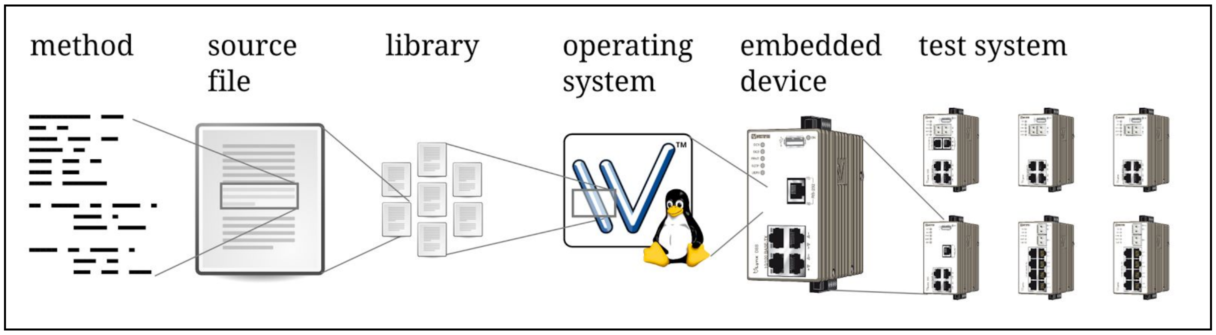

FIGURE 1. Overview of testing structure from source code to test system.

An automated test case is implemented as a Python class, which includes requirements on the test system on how many DUTs the test needs and how they are interconnected. Instructions in the test cases specify how the individual DUTs should be configured. 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 numerous individual test cases. Depending on the number of WeOS versions to test, and the types of suites needed, between one and ten test suites are required per test system per night.

2.1. Challenge 1: Nightly Testing does not Finish on Time

Tests on target devices can be lengthy. For example, if a DUT needs to reboot, or if the DUT includes special ports, the device may need time to load the firmware before it reaches the desired state. Tests of common ethernet protocols may also have intrinsic time-consuming characteristics.

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 12 seconds to 37 minutes, with an average of between 1.6 and 2.5 minutes.

The nightly test suites of WeOS grew over time with the implementation of more functionality and test cases. Eventually, the nightly suites did not finish until after 8am, when the Westermo work day starts. When that happened, we sometimes manually stopped the suite's execution, which required manual labor and sometimes led to undefined states in the test systems.

2.2. Challenge 2: Manual Work and Forgotten Tests

When test suites ran too long, Westermo tried to implement a workaround by manually removing tests. The plan was to manually reinsert these tests during the weekend when there was more time for testing. This often led to tests being ignored for months, resulting in failure to detect issues in the operating system. Moreover, any manual work with the test suites was risky as one might unintentionally introduce syntax errors, or break the suite in other ways.

2.3. Challenge 3: Unprioritized Test Cases

Tests are typically named according to the functional area they test and were ordered alphabetically in the regression suites. If a test had a name late in the alphabet, it was executed near the end of the test run and was therefore frequently canceled due to lack of time. If they were run but failed, the failure would be noted late in the suite, providing less rapid feedback and less time for debugging.

The alphabetical ordering also concealed failures that were previously unknown and that we discovered when the suites were manually reordered.[5,6] These failures indicated insufficient clean up in various state transitions of the test framework or in WeOS.

3. The SuiteBuilder Tool

To address the challenges, Westermo designed SuiteBuilder,[1] an automated tool that assigns a vector of independent priorities to each test case, and then merges the priorities into a single final value that is used as a basis for ordering the entire regression suite. Some priorities are assigned statically by humans, while others are derived automatically from historical test outcomes or source code changes. The priority merging process is controlled by weights for each priority that can be adjusted by the test team, according to the goals deemed most important at a particular time. Figure 2 shows the role of SuiteBuilder in the regression testing process.

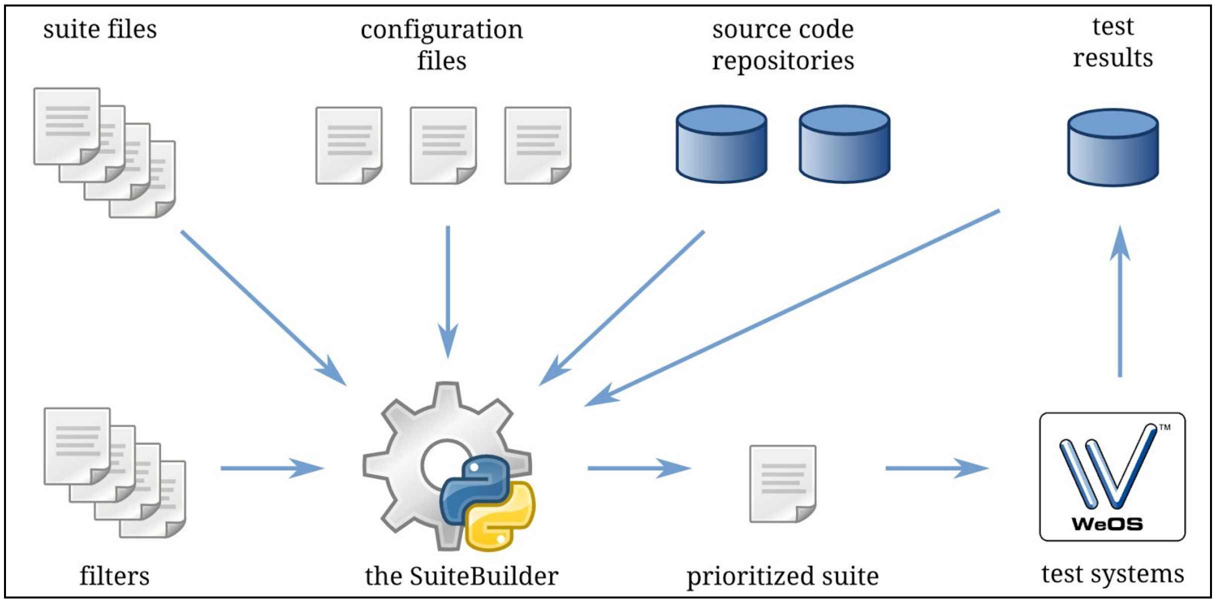

FIGURE 2. Overview of the SuiteBuilder Tool.

The following 5 steps describe SuiteBuilder's process of creating and executing a regression test suite.[1]

1. **Collection of raw test suites.** SuiteBuilder parses the available tests, and builds a *raw test suite.*

2. **Assigning priorities.** Each individual *prioritizer* aims at a specific desirable characteristic, and can assign one of the individual priority variables LOWEST, LOW, MEDIUM, HIGH or HIGHEST to a test case in the raw suite. SuiteBuilder users define specific numeric values for each variable; a typical set of values is 1,3,5,7,10.
    a. The *TestPrioritizer* assigns a static prioritization for individual tests, assigning an increased priority for tests that can only run on a few test systems, and a decreased value for tests that can be run on many test systems. Experienced WeOS developers write these values into configuration files on a case-by-case basis.
    b. The *TagPrioritizer* assigns a prioritization for an entire group of tests that share a common *tag*. WeOS developers organize the tests into groups and assign tags based on domain and test framework knowledge. The TagPrioritizer provides a coarse-grained prioritization for the group, while the TestPrioritizer provides a fine-tuned priority for individual tests.
    c. Failing tests might indicate issues in WeOS, so the *FailPrioritizer* increases the priority of a test that has failed recently or frequently. A test that failed within the past 3 days is assigned a high value. Failures that occurred longer ago receive correspondingly lower priorities. Generally, the smaller the number of times a test case has failed, the lower the assigned priority, to avoid rerunning tests that consistently pass.
    d. The *RecentPrioritizer* increases priority for tests that have not been selected recently. This enhances test circulation and assures that all tests are periodically executed. It also "demotes" tests that have been executed repeatedly without recently failing.
    e. *SourcePrioritizers* assign priorities based on changes to source code. The source code repositories of WeOS and the test framework provide detailed logs describing where code changes were made either in the software or in the test framework. Two *SourcePrioritizers* (one each for WeOS and the test framework) search these logs for terms that match test group tags, and increase the priority of tests associated with code areas that have recent changes. A typical entry in the change log might include a text comment like "Fixed bug 73 in poe wrapper", and list the files that were edited. Such an entry causes the SourcePrioritizer to increase the priority of all tests in the group with the tag `poe`. Like the Fail and RecentPrioritizers, the SourcePrioritizers set their priority based on how far back in time they need to search. For example, the highest priority is assigned to a tag that has received changes within the previous day.

3. **Merging priorities.** The *PriorityMerger* gives each test case a final overall priority by using a weighted average of the individual priorities that have been assigned.[7,8] Note that some prioritizers may not be relevant to a given test case, and therefore would not assign a value to that case. The merging step produces an ordered list of all test cases in the raw test suite, sorted based on their computed final priority.

   The test team can decide on the merging weights depending on the relative importance of the different prioritizers to the software development organization, as well as the organization's past testing experience. Westermo gave the FailPrioritizer the largest weight of 9, as their experience was that recently failing tests were the most important. The other assigned weights might be 6 for both SourcePrioritizers, 4 for the RecentPrioritizer, 3 for the TestPrioritizer, and 2 for the

TagPrioritizer. Of course, the weights can be changed as needed for different types of systems or test libraries.

4. **Constructing the suite.** SuiteBuilder assigns a *desired duration* of execution time for the final suite, typically a portion of the full night. The tool then uses the priority and expected running time of each test case to determine whether and when it places the test into the suite. SuiteBuilder first removes any test cases from the raw suite that the target test system's hardware cannot execute. Next it filters out experimental tests, tests for experimental functional areas, and tests that trigger known issues. Experience has shown that these tests sometimes drive the test framework into undefined, unrecoverable states, and cause remaining test results to be lost. SuiteBuilder places the remaining test cases into the suite in order of priority, as long as the tests already in the suite have not exhausted its desired duration. The *expected duration* of the final suite is the sum of the expected times of all its test cases. A test case's expected time is based on its prior running time, or has a default of 3 minutes if it has no prior history. Since the expected time of the last test case added can be longer than the suite's remaining time, the expected duration of the final suite can be longer than its desired duration.

5. **Executing the suite.** The test framework executes the final suite, and stores the results from the test session in the test results database.

4. SuiteBuilder Tool Implementation

The entire SuiteBuilder solution is 15 modules of Python totalling a few thousand lines of code. The primary building blocks of SuiteBuilder are the test result database, tags, prioritizers and the suite handler library.[1]

4.1. Test Result Database

At Westermo, we have designed a relational database that keeps millions of test execution results. This database provides one of the major sources of data for the prioritizers and 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 of a run of a particular test case?"

4.2. 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. Testers create the tags manually 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. A manually built configuration file records each tag and the patterns that identify the tests with that tag. This is an example of a tag definition for `poe`.

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

4.3. Prioritizers

All prioritizers are sub-classes of a common class *BasePrioritizer.* The base class handles information on test cases, and can be asked the current priority of a test case. A sub-class prioritizer can either *increase,* or *decrease,* or *leave unchanged* the priority of a test case.

This example illustrates how to extend the base class 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) query the relevant configuration files in order to determine their priorities. In contrast, FailPrioritizer and RecentPrioritizer query the Test Results Database to obtain their priorities.

4.4. 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 information is recorded in a set of configuration files that are maintained in a *Suite Handler Library*.

5. Evaluating SuiteBuilder

SuiteBuilder was run on Westermo's test systems to assess its effectiveness, and assure that it met the challenges discussed above.

5.1. Challenge 1: Nightly Testing Does Not Finish on Time

The most important issue addressed by SuiteBuilder was the failure of test suites to finish execution within the time allocated for nightly regression testing. SuiteBuilder addresses this issue by constructing suites with an expected duration that is less than or only slightly above the total available nightly time, so that they do not have to be terminated at the start of the workday because they had not run to completion. We evaluated the success of this strategy by comparing the actual running time durations of 701 test suites created by SuiteBuilder to the expected durations of those suites.[1]

Figure 3 shows that the running time of almost all suites was very close to their expected durations. For some suites, the actual time was significantly lower than the expected time (red dots far below the blue curve). These were typically suites for which no test cases could be found, suites that crashed, were aborted or that had poor estimates, perhaps because of one of the above issues in the past.

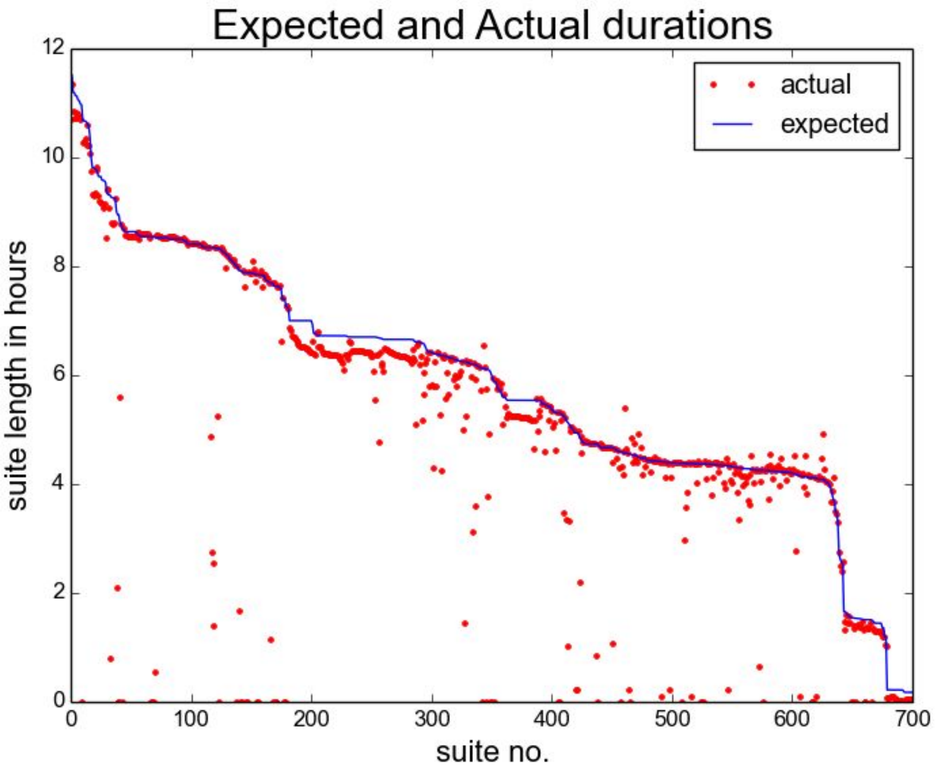

FIGURE 3. Actual and expected durations of suites.

No suite ran for more than 120% of its expected duration. The largest underestimation was 50 minutes. Of the 701 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,
- 45 had no duration at all. These were either empty or aborted at the start.

This evaluation indicates that test suites now rarely exceed their expected time and therefore typically run to completion without having to be aborted at the start of business the following day.

5.2. Challenge 2: Manual Work and Forgotten Tests

To investigate whether SuiteBuilder omitted any important tests, we collected all tests that had been part of regular nightly testing for nine months preceding introduction of SuiteBuilder and investigated whether any of them were left out of SuiteBuilder suites in its first two months of use.[1]

SuiteBuilder generated all meaningful tests that had existed in the preceding nine months. In some cases, the generated tests differed only in having parameters in a different order, or having different names from the original tests. We conclude that the tool would not allow any test to be systematically omitted for months at a time.

Experienced test specialists who manually worked with the suites were generally pleased with SuiteBuilder since it addressed the problems they had faced:

- they no longer had to manually alter suites to provide more testing during weekends
- removed or rescheduled tests were not forgotten
- the lengths of test suites were dynamically changed based on the available time for testing
- SuiteBuilder-generated suites usually finished on time, so partially completed suites did not have to be aborted each morning
- particularly important tests were not left unexecuted.

5.3. Challenge 3: Unprioritized Test Cases

The former alphabetical ordering of test cases gave no precedence to tests that were most likely to detect faults. To determine if SuiteBuilder constructed suites that provided early detection of faults, we considered test suites that contained at least 20 test cases, and did not lead either the WeOS or the test framework into an unstable state. We compared the distribution of failing tests in the 6758 suites that ran during the 23 months before the introduction of SuiteBuilder against the distribution of failures in the 8930 suites run during the 23 months after its introduction.[1]

Since the suites are typically not of the same length, we scaled the index of each failing test to a number between 0 and 1, so that the normalized position could be expressed as $p_{norm} = p/len$, where p represents the test's position in the suite and len is the number of tests in the suite.

Figure 4 shows the failure distributions of the pre- and post-SuiteBuilder tests. Each test suite is divided into 20 segments containing equal numbers of test cases in order of their execution. The height of the *ith* bar represents the percentage of all fault-detecting test cases that ran in the *ith* segment of all the suites over the entire 23 month period. Thus the first bar represents the percentage of failures that were detected by the first 5% of test cases that ran in all suites during the relevant 23 month period.

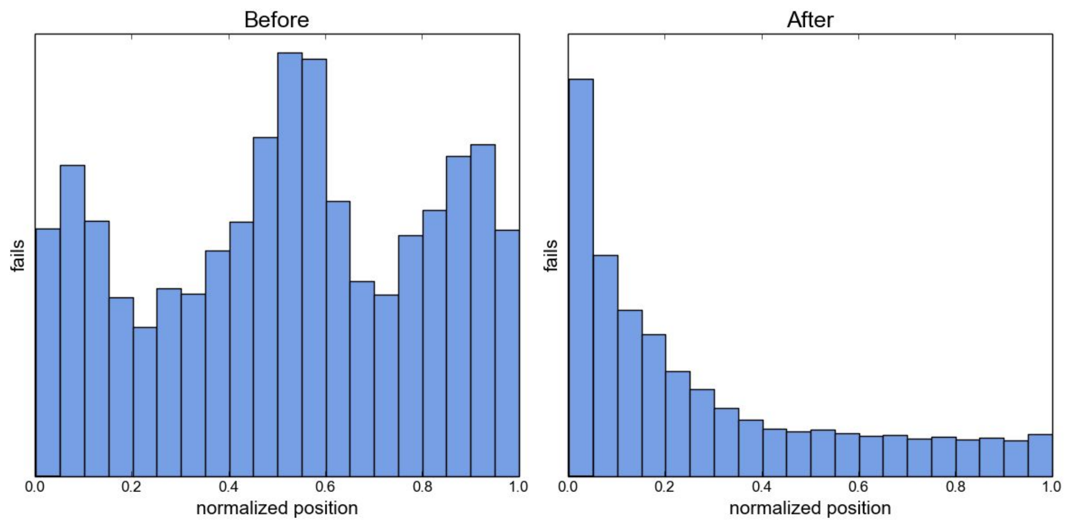

FIGURE 4. Normalized failure distributions before and after SuiteBuilder.

The plots show that fault detection is highest in the earliest tests in the post-SuiteBuilder suites, in contrast to the fluctuating pattern of detection by tests in the pre-SuiteBuilder suites. Before SuiteBuilder was introduced, tests in the first third of the suites detected only about 27% of the failures. In the 23-month period after the introduction of SuiteBuilder, the first third of the tests revealed more than 65% of the failures.

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.

Conclusions

We placed SuiteBuilder in production in the late spring of 2014. Since then it has been used continuously to build suites for nightly testing. SuiteBuilder solves real problems in an ecosystem of build servers, test systems with target devices, source code management systems, a test result database, and human beings. It rapidly builds suites for many test systems, and the time required to build the suites is small compared to the time needed to run the test cases.

We made a number of significant changes as we gained experience:

- We reconfigured the PriorityMerger weights to increase the impact of the FailPrioritizer.
- Indices in the database tables and other optimizations in the database queries were introduced to speed up queries.
- The SourcePrioritizer for the test framework was introduced.

While the SuiteBuilder has significantly improved both failure detection and the overall regression testing process, we continue to investigate ways of improving its operation. For example, an experimental new prioritizer was recently implemented that places all tests of a particular topic first in the suite. This is sometimes useful to allow testers to investigate possible unusual test case performance. We also speculate that the tool might miss certain test cases, e.g., when a code branch is given too little testing time. In such a situation, we have no reason to assume that every test case is executed with at least a desired minimal frequency. Another topic that deserves additional research is a more systematic, optimized assignment of priority weights, although the manual assignment of prioritizers' weights has worked well until now.

Acknowledgements

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. P. E. Strandberg et al., “Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors,” Proc. 27th Int'l Symp. Software Reliability Engineering (ISSRE16), 2016, pp. 12–23.
2. S. Elbaum et al., “Techniques For Improving Regression Testing in Continuous Integration Development Environments,” Proc. 22nd Int'l Symp. Foundations of Software Engineering (FSE 14), 2014, pp. 235–245.
3. H. Hemmati et al., “Prioritizing Manual Test Cases in Traditional and Rapid Release Environments,” Proc. 8th Int'l Conf. Software Testing, Verification and Validation (ICST 15), 2015, pp. 1–10.
4. S. Yoo and M. Harman, “Regression Testing Minimization, Selection and Prioritization: A Survey,” *Software Testing, Verification and Reliability,* vol. 22, no. 02, 2012, pp. 67–120.

5. Q. Luo et al., “An Empirical Analysis of Flaky Tests,” Proc. *22nd Int’l Symp. Foundations of Software Engineering (FSE 14)*, 2014, pp. 643–653.
6. S. Zhang et al., “Empirically Revisiting the Test Independence Assumption,” *Proc. 2014 Int’l Symp. Software Testing and Analysis (ISSTA 14)*, 2014, pp. 385–396.
7. E. Engström et al., “Improving Regression Testing Transparency and Efficiency With History-based Prioritization–An Industrial Case Study,” *Proc. 5th Int’l Conf. Software Testing, Verification and Validation (ICST 2011)*, 2011, pp. 367–376.
8. Y. Fazlalizadeh et al., “Prioritizing Test Cases for Resource Constraint Environments Using Historical Test Case Performance Data”, *Proc. 2nd Int’l Conf. Computer Science and Information Technology (ICCSIT 2009)*, 2009, pp. 190–195.
9. European Association of Research & Technology Organizations, “The TRL Scale as a Research and Innovation Policy Tool, EARTO Recommendations,” 2014, pp. 1–17.