Test Suite Parallelization in Open-Source Projects: a Study on its Usage and Impact

[[Authors omitted for review]]

***Abstract*—Dealing with high testing costs remains an important problem in Software Engineering. This paper reports our findings on the usage and impact of test execution parallelization in open-source projects. Cost-effective multicore CPUs, as well as testing frameworks and build systems to capitalize on them, are widely available today.**

Considering a set of 390 popular Java projects we analyzed, we found that 21% of the projects contain costly test suites but parallelization features still seem underutilized in practice — only 18.1% of costly projects use parallelization. The main reported reason for adoption resistance was the concern to deal with concurrency issues. Results suggest that, on average, developers prefer high predictability than high performance in running tests.

This study brings to light the benefits and burdens of test suite parallelization to improve test efficiency. It provides recommendations to practicioners and developers of new techniques and tools aiming to speed up test execution with parallelization.

I. Introduction

As software evolves it is expected that the number of tests and the length of test runs increase. Both components can add up to the aggregate cost of running a test suite. Dealing with high testing costs is an important problem in software engineering research and industrial practice.

Several approaches have been proposed in the research literature to address the regression testing problem, with the focus mainly on test suite minimization, prioritization, reduction, and selection [27]. In industry, the focus has been mainly on distributing the testing workload. Evidence of this are the Google TAP system [10], [26] and the Microsoft CloudBuild system [21], which provide distributed infrastructures to efficiently build massive amounts of code and run tests. Building in-house server clusters is also a popular mechanism to distribute testing workloads. For example, as of August 2013, the test suite of the Groupon PWA system, which powers the groupon.com website, included over 19K tests. To run all those tests under 10m, Groupon used a cluster of 4 computers with 24 cores each [14].

At large organizations, the alternative of renting cloud services[1] or even building proprietary infrastructures for running tests is a legitimate approach to mitigate the regression testing problem. However, for projects with modest or nonexistent budgets and yet relatively heavy testing workloads, this solution may not be economically viable. For these cases, the use of commodity hardware (e.g., existing workstations) is an attractive solution for running tests. The proliferation of multi-core CPUs and the increasing popularization of testing frameworks and build systems, which today provide mature support for parallelization, enable speedups through increased CPU usage (see Section II). These two elements — demand for cost-effective test execution and supply of relatively inexpensive testing infrastructures — led us to investigate the usage of parallelism to speed up test execution.

This paper reports on an empirical study we conducted to analyze the usage and impact of low-level parallelization to speed up testing in open-source projects. This is a relevant problem given the tremendous popularity of open-source development and regression testing research [27]. Note that parallelization is complementary to other approaches to mitigate testing costs such as (safe) test selection [8], [22] and continuous integration [23].

The dimensions of analysis we considered in this study are (i) feasibility, (ii) adoption, (iii) speedup, and (iv) tradeoffs. The dimension *feasibility* measures the potential of parallelization to reduce testing costs. In the limit, parallelization would be fruitless if all projects used short-running test suites or if the execution cost was dominated by a single test case in the suite. The dimension *adoption* evaluates how often existing open-source projects use parallelization schemes and how developers involved in costly projects (not using test suite parallelization) perceive this technology. It is important to measure resistance of practitioners to the technology and to understand their reasons. The dimension *speedups* evaluates the observed impact of parallelization in running times. Finally, the dimension *tradeoffs* considers the problems in running tests in parallel, including test flakiness [15]. We briefly summarize our findings in the following.

Feasibility. To assess how prevalent long-running test suites are we selected 390 popular Java projects from Github containing Maven build files [17]. Section IV details our methodology to select subjects and to isolate our experiments from environmental noise. Results indicate that nearly 21% of the projects take at least 1m to run their test suites and 8.7% of the projects take at least 5m to run their suites. Considering the 83 projects in those two groups, the average execution time of a test suite was 8m36s. Results also show that test cases are typically short-running, typically taking less than half a second to run. Furthermore, we found that only in rare cases few test cases monopolize execution cost associated with each test suite.

Adoption. We considered two aspects in measuring technology

[1]List of popular hosting cloud services: https://clutch.co/cloud

adoption. First, we measured usage of parallelism in open-source projects. Then, we ran a survey with developers to understand the reasons that could explain resistance to using the technology. Considering only those projects whose test suites that longer than a minute to run, we found that only 13.3% of them use parallelism. When correcting this value with the feedback obtained a posteriori from developers, we realized that the proportion should be higher, of 18.1%. For the qualitative analysis we surveyed developers from a selection of costly projects that did not use parallelization. Results indicate that test suite parallelization is underutilized. Dealing with concurrency-related issues (e.g., the extra work to organize test suite to avoid concurrency errors) and the availability of continuous integration services were the most frequently answered reasons for not considering parallelization.

Speedups. We used two setups to measure execution speedups. In one setup we measured the speedups obtained on projects that we knew a priori ran their test suites in parallel by default. In the other setup, we evaluated how execution scales with the number of available cores in the machine. Considering the first setup, results indicate that the average speedup of parallelization was 4.4x. Although we found cases with very high speedups (e.g., 28.8x for project Jcabi), we found cases where speedups were not very significant. Considering the scalability experiment, we noticed, perhaps as expected, that speedups are bounded by long-running test classes.

Tradeoffs. Test flakiness is a central issue that can arise when running tests in parallel. Dependent tests can be affected by different schedulings of test methods and classes. This dimension of the study measures the impact of different parallel configurations on test flakiness and speedup. Overall, results indicate that parallel configurations that fork JVMs do not achieve speedups as high as configurations that parallelize test classes and methods, however they manifest much lower flakiness ratios.

Our observations may trigger multiple actions:

- *Incentivize forking.* Forked JVMs manifest very low rates of test flakiness. Developers of projects with long-running test suites should consider using that feature, which is available in modern build systems today (e.g., Maven).
- *Break test dependencies.* Non-forked JVMs can achieve impressive speedups at the expense of sometimes impressive rates of flakiness. Breaking test dependencies to avoid flakiness and take full advantage of those options is advised for developers with greater interest in efficiency.
- *Refactor tests for load balancing.* Forked JVMs scales better with the number of cores when the test workload is balanced across testing classes. To balance the workload, automated user-oblivious refactoring can help in scenarios where developers are not willing to change test code but have access to machines with a high number of cores.
- *Improve debugging for build systems.* While preparing our experiments, we found scenarios where Maven's executions did not reflect corresponding JUnit's executions. (Docker reproduction scripts available.) Those issues can hinder developers from using parallel testing. Better debugging infrastructure is important.

The artifacts we produced as result of this study are available from the following web page https://doubleblind.000webhostapp.com/.

II. Parallel Execution of Test Suites

Figure 1 illustrates the different levels where parallelism in test execution can be obtained. The highest level indicates parallelism obtained through different machines on the network. For instance, using virtual machines from a cloud service to distribute test execution. The lowest levels denote parallelism obtained within a single machine. These levels are complementary: the lowest levels leverage the computing power of server nodes whereas the highest level leverages the aggregate processing power of the farm.

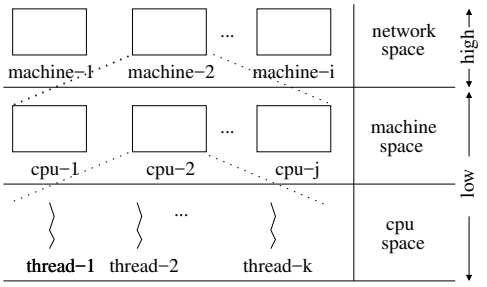


Figure 1: Levels of parallelism.

This paper focuses on low-level parallelism, where computation can be offloaded at different CPUs within a machine and at different threads within each CPU. This form of parallelism is enabled through build systems (spawning processes in different CPUs) and testing frameworks (spawning threads in one given CPU). It is important to note that a variety of testing frameworks provide today support for parallel test execution (e.g., JUnit [11], TestNG [25], and NUnit [18]) as to benefit from the available power of popular multi-core processors. In the following, we elaborate relevant features of testing frameworks and build systems. We focused on Java, Maven, and JUnit but the discussion can be generalized to other language and tools.

A. Testing Frameworks

The list below shows the choices to control parallelism within one Java Virtual Machine (JVM). These options are offered by the testing framework (e.g., JUnit).

- **Sequential (C0).** No parallelism is involved.
- **Sequential classes; parallel methods (C1).** This configuration corresponds to running test classes sequentially, but running test methods from those classes concurrently.
- **Parallel classes; sequential methods (C2).** This configuration corresponds to running test classes concurrently, but running test methods sequentially.
- **Parallel classes; Parallel methods (C3).** This configuration corresponds to the combination of C1 and C2.

Notice that an important aspect in deciding which configuration to use (or in designing new test suites) is the possibility of race conditions on shared data during execution. Data sharing can occur through state that is reachable from statically-declared variables in the program or through variables declared within the scope of the test class [15]. Considering data race avoidance, configuration C1 is preferable over C2 when it is clear that test methods in a class do not manipulate shared state, which can be challenging to determine [4]. Similarly, C2 is preferable over C1 when it is clear that several test methods in a class perform operations involving shared data. Configuration C3 does not restrict scheduling orderings. Consequently, it is more likely to manifest data races during execution. Note that speedups depend on several factors, including the test suite size and distribution of test methods per class.

B. Build Systems

Forking OS processes to run test jobs is the basic mechanism of build systems to obtain parallelism at the machine space (see Figure 1). For Java-based build systems, such as Maven and Ant, this amounts to spawning one JVM, on a given CPU, to handle a test job and aggregating results when jobs finish. The list below shows the choices to control parallelism through the build system (e.g., Maven).

- **Forked JVMs with sequential methods (FC0).** The build system spawns multiple JVMs with this configuration, assigning a partition of the set of test classes to each JVM. Test methods run sequentially within each JVM.
- **Forked JVMs with parallel methods (FC1).** With this configuration, the build system forks multiple JVMs, as FC0 does. However, it runs tests concurrently, as C1 does.

Note from the listing that forking can only be combined with configuration C1 (see Section II-A) as Maven made the design choice to only accept one test class at a time per forked process. Maven offers an option to reuse JVMs that can be used to attenuate the potentially high cost of spawning new JVM processes on every test class (if reuse is enabled) and also to achieve test isolation (if reuse is disabled).

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <forkCount>1C</forkCount>
        <reuseForks>true</reuseForks>
        <parallel>methods</parallel>
        <threadCount>5</threadCount>
    </configuration>
</plugin>
```

Figure 2: Configuration FC1 on Maven.

Example: Figure 2 shows a fragment of a Maven configuration file, known as *pom.xml*, highlighting options to run tests using the parallel execution mode FC1. Maven implements this feature through its Surefire JUnit test plugin [20]. With this configuration, Maven forks one JVM per core (`forkCount` parameter) and uses five threads (`threadCount` parameter) to run test methods (`parallel` parameter) within each forked JVM. Maven reuses created JVMs on subsequent forks when execution of a test class terminates (`reuseFork` parameter).

III. Objects of Analysis

We used Github's search API [7] to identify projects that satisfy the following criteria: (1) the primary language is Java[2], (2) the project has at least 100 stars, (3) the latest update was on or after January 1st, 2016, and (4) the *readme* file contains the string *mvn*. We focused on Java for its popularity. Although there is no clearcut limit on the number of Github stars [6] to define relevant projects, we observed that 100 was enough to eliminate trivial subjects. The third criteria serves to skip projects without recent activity. The fourth criteria is an approximation to find Maven projects. We focused on Maven for its popularity on Java projects. Important to highlight that, as of now, the Github's search API can only reflect contents from README file (not other code elements); it does not provide a feature to search for projects containing certain files in the dir structure (e.g., *pom.xml*). Figure 3 illustrates the query to the Github API as an HTTP request.

```
https://api.github.com/search/repositories?q=language:java
        +stars:>=100+pushed:>=2016-01-01
        +mvn%20in:readme+sort:stars
```

Figure 3: Query to the Github API for projects with the following criteria: (1) Java, (2) at least 100 stars, (3) updated on January 1st, 2016 (or later), (4) contains the string *mvn* in the *readme* file. Output is paginated in descending order of stars.

After obtaining a list of potential projects, we filtered those containing a *pom.xml* file in the root directory. A Maven project may contain several sub-modules with multiple *pom.xml* files. As of March 25th 2017, our search criteria returned 685 subjects. Figure 4 summarizes our sample set.

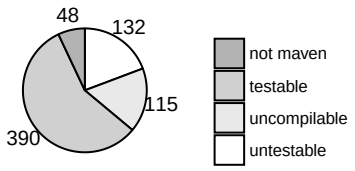


Figure 4: We fetched 685 popular projects hosted on Github. From this initial sample, we ignored 48 projects without Maven support, 115 uncompilable projects, and 132 untestable projects. We considered 390 projects to conduct our study.

From 685 downloaded projects, 48 projects were not Maven or did not have a *pom.xml* in the root directory, 115 were unable to compile due to missing dependencies, and 132 projects were untestable due to incompatible testing environment. We also found 16 projects with "flaky" tests. We considered those projects but ignored their tests (using JUnit's `@Ignore` annotation) as to avoid measurement noise. To detect those projects, we executed every test suite for three times. Our final set of subjects consists of 390 projects.

[2]In case of projects in multiple languages, the Github API considers the predominant language as the primary language.

A. Setup and Replication

To run our experiments, we used a Core i7-4790 (3.60 GHz) Intel processor machine with eight virtual CPUs (four cores with two native threads each) and 16GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). Software settings include git to fetch subjects, Java 8, and Maven 3.3.9 to build and test subjects. We used Python, Bash, R and Ruby to process the data and generate plots. All source artifacts are publicly available for replication (on our website [2]). This includes supporting scripts (e.g., the script that test subjects and generates raw analysis data) and the full list of projects.

IV. Evaluation

We pose the following research questions, organized by the dimensions of analysis we presented in Section I.

- Feasibility
  - **RQ1.** How prevalent is the occurence of time-consuming test suites?
  - **RQ2.** How time is distributed across test cases?
- Adoption
  - **RQ3.** How popular is test suite parallelization?
  - **RQ4.** What are the main reasons that prevent developers from using test suite parallelization?
- Speedups
  - **RQ5.** What are the speedups obtained with parallelization (in projects that actually use it)?
  - **RQ6.** How test execution scales with the number of available CPUs?
- Tradeoffs
  - **RQ7.** How parallel execution configurations affect testing costs and flakiness?

A. Feasibility

- RQ1. **How prevalent is the occurence of time-consuming test suites?**

To evaluate prevalence of projects with costly test suites, we considered the 390 testable subjects from Figure 4. Figure 5 illustrates the script we used to measure time.

We took the following actions to isolate our environment from measurement noise. First, we observed that some test tasks called test-unrelated tasks (e.g., *javadoc* generation and static analyses) that could interfere in our time measurements. To address that potential issue, we inspected Maven execution logs from a sample including a hundred projects prior to running the script from Figure 5. The tasks we found were ignored from execution (lines 1-3). Furthermore, to avoid noise from operating system events, we configured our workstation to run only essential services. The machine was dedicated to our experiments and we accessed it via SSH. In addition, we configured the `isolcpus` option from the Linux Kernel [13] to isolate six virtual CPUs to run our experiments, leaving the remaining CPUs to run OS processes [5]. The rationale for this decision is to prevent context-switching between user processes (running the experiment) and OS-related processes. Finally, to make sure our measurements were fair, we compared timings corresponding to the sequential execution of tests using Maven with that obtained with JUnit's default `JUnitCore` runner, invoked from the command line. Results were very close.

The main loop (lines 5-11) of the script in Figure 5 iterates over the list of subjects and invokes Maven multiple times (lines 7-9). It first compiles the source and test files (line 7), make all dependencies available locally (line 8), and then runs the tests in offline mode as to skip the package update task, enabled by default (line 9). After execution, we used a regular expression on the output log to extract elapsed time (line 10).

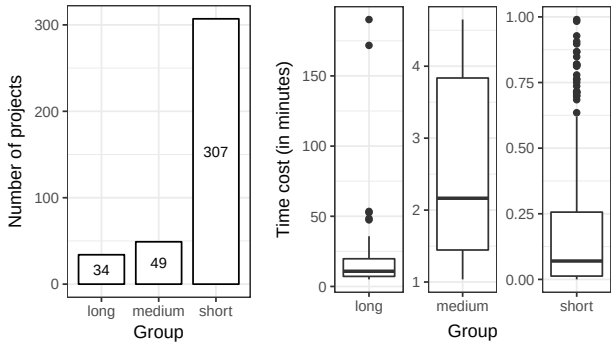
```
1  MAVEN_IGNORES="-Dmaven.javadoc.skip=true -Drat.skip=true \
2      -Djacoco.skip=true -Dcheckstyle.skip=true \
3      -Dfindbugs=true -Dcobertura.skip=true"
4
5  for subj in $SUBJECTS; do
6      cd $SUBJECTS_HOME/$subj
7      mvn clean install -DskipTests $MAVEN_IGNORES
8      mvn dependency:go-offline
9      mvn -o test $MAVEN_IGNORES &> $LOG
10     cat $LOG | grep "\[INFO\] Total time:"
11 done
```

Figure 5: Bash script to measure time cost of test suites. For each subject, we compile the source and test files, fetch all dependencies, and execute the tests in offline mode ignoring non-related tasks. Test-unrelated tasks are omitted.

We ran the test suite for each subject three times, reporting averaged execution times in three ranges: tests that run within a minute (short group), tests that run in one to five minutes (medium group), and tests that run in five or more minutes (long group). We followed a similar methodology to group projects by time as Gligoric et al. [8] in their work on regression test selection. Figure 6a shows the number of projects in each group. As expected, long and medium projects do not occur as frequently as short projects. However, they do occur in relatively high numbers.

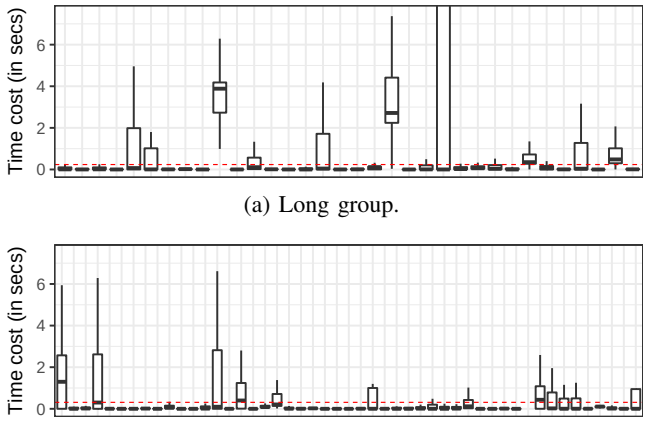
Figure 6b shows cost distribution of test suites in each group as boxplots. Note that the y-ranges are different. The distribution associated with the short group is the most unbalanced (right skewed). The test suites in this group ran in 15 or less seconds for over 75% of the cases. Such scenarios constitute the majority of the cases we analyzed. Considering the groups medium and long, however, we found many costly executions. Nearly 75% of the projects from the medium group take over 3.5 minutes to run and nearly 75% of the projects from the long group take $\sim$20 minutes to run. We found cases in the long group were execution takes over 50 minutes to complete, as can be observed from the outliers (dots) in the plot.

It is important to note that we under-estimated cost in our experiments for two main reasons. First, some tests may finish earlier than expected due to the observed test failures in some of the revisions we downloaded. From the 390 testable projects, 250 successfully executed all tests and 140 reported some test failures. Second, some projects may omit long-running tests on their default execution. For instance, the project `apache.maven-surefire` runs all unit tests in a few seconds. According to our criteria, this project is to



(a) (b)

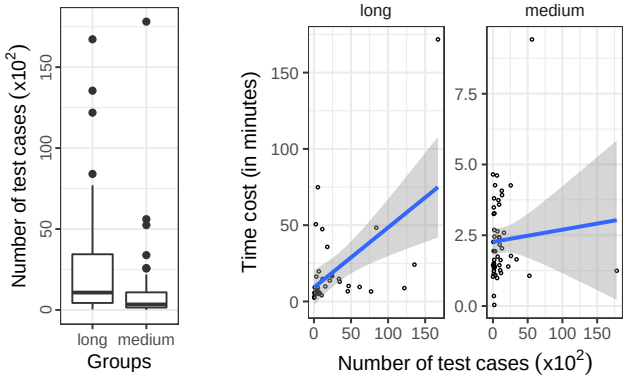
Figure 6: (a) Subjects grouped by time cost (t): short run ($t < 1m$), medium run ($1m \leq t < 5m$), and long run ($5m \leq t$); (b) Distribution of time cost by group.



(a) Long group.

(b) Medium group.

Figure 7: Time distributions.



(a) Size of test suites. (b) Size versus running time of test suites.

Figure 8: Relating size and time.

be classified as short but a closer look reveals that only smoke tests are run by default in this project. Time-consuming integration and system tests are only accessible via custom parameters, which we do not handle in our experimental setup. We enabled such parameters for this specific project and observed that testing time goes to nearly 30 minutes. For simplicity, we considered only the tests executed by default.

Answering RQ1: We conclude that time-consuming test suites are relatively frequent in open-source projects. We found that 21% of the 390 projects we analyzed (i.e., over 1 in every 5 projects) take at least 3 minutes to run and 8.7% take at least 5 minutes to run.

- RQ2. **How time is distributed across test cases?**

Section IV-A showed that medium and long-running projects are not uncommon, accounting to nearly 21% of the 390 projects we analyzed. Research question RQ2 measures the distribution of test costs in test suites as to estimate (lack of) potential of obtaining speedups with parallelization. In the limit, if cost is dominated by a single test from a large test suite, it is unlikely that parallelization will be beneficial as a test method is the smallest working unit in test frameworks.

Figures 7a and 7b show the time distribution of individual test cases per project. We observed that the average median value of execution cost for a test was relatively small (dashed horizontal red lines), namely 0.31s for medium projects and 0.23s for long projects. The standard deviations associated with each distribution were relatively low. We noted a small number of cases of CPU monopolization. For example, the highest value of σ occurred in `uber_chaperone`, a project from the medium group. This project contains only 65 tests, 62 of which take less than 0.5s to run, one of which takes nearly 3s to run, and two of which take ∼40m to run. For this project, 99.2% of the execution cost is dominated by only 3% of the tests; without these two costly tests this project would have been classified as short-running. A closer inspection in the data indicates that the project `uber_chaperone` was a corner case; we did not find other projects with such extreme time monopolization profile. Project `facebookarchive_linkbench` is also classified as long-running and has the second highest value of σ. For this project, however, cost is distributed more smoothly across 98 tests, of which 8 (8.1%) take more than 1s to run with the rest of the tests running faster.

Figure 7 showed that the average median times were similar for medium and long-running test suites. Results indicate that the difference in overall running times of projects in those groups was mainly justified by the number of test cases as opposed to the individual costs of test cases. Figure 8a shows the difference in the distribution of test suite sizes across groups. This figure indicates that long projects, albeit having a wider inter-quartile range (middle 50% projects in this group are less predictable), have a higher median and much higher average number of test cases. Furthermore, we noted a strong positive correlation between running time and number of test on projects in the long group. Considering the medium group, the correlation between these two variables was weak. Figure 8b illustrates this correlation.

Answering RQ2: Overall, results indicate that projects with a very small number of tests monopolizing end-to-end execution time were rare.

B. Adoption

The dimension adoption focuses on the usage of parallelism in open-source projects. It evaluates (RQ3) how often open-source projects use parallelization schemes and (RQ4) how developers involved in costly projects, not using parallelization, perceive this technology.

- RQ3. **How popular is test suite parallelization?**

To answer RQ3, we selected all projects from the medium and long groups, i.e., projects that ran in at least one minute. This set includes 83 projects (see Section IV-A). We looked for dynamic and static manifestations of parallelism.

1) Dynamic checking: To find dynamic evidence of parallelism, we ran the test suites from our set of 83 projects to output all key-value pairs of Maven parameters. To that end, we used the option `-X` to produce debug output and the option `-DskipTests` to skip execution of tests. We skipped execution of tests as we observed from sampling that only bootstrapping the Maven process suffices to infer which parallel configuration modes it will use to actually run the tests. It is also important to point that we used the default configurations specified in the project. We inferred parallel configurations by searching for certain configuration parameters in log files. According to Maven's documentation [20], a parallel configuration depends either on (1) the parameter `parallel` to define the parallelism mode within a JVM followed by the parameter `threadCount` or (2) the parameter `forkCount`[3] to define the number of forked JVMs. As such, we captured, for each project, all related key-value pairs of Maven parameters and mapped those pairs to one of the possible parallelization modes. For instance, if a given project contains a module with the parameter `<forkCount>1C</forkCount>`, the possible classifications are FC0 or FC1, depending on the presence and the value of the parameter `parallel`. If the parameter `parallel` is set to `methods` the detected mode will be FC1. Large projects may contain several test suites distributed on different Maven modules potentially using different configurations. For those cases, we collected the Maven output from each module discarding duplicates as to avoid inflating counts for configuration modes that appear in several modules of the same project. For instance, if a project contains two modules using the same configuration, we counted only one occurrence.

Figure 9 shows the projects we identified where parallelism is enabled by default in Maven. Column "*Subject*" indicates the name of the project, column "*# of modules*" indicates the fraction of modules containing tests that use the configuration of parallelism mentioned in column "*Mode*". We note that, considering these projects, the modules that do not use the configuration cited use the sequential configuration C0. For example, six modules (=31-25) from Log4J2 use sequential configuration.

Subject	*# of modules*	*Mode*
Chaos	1/1	C2
Flink	74/74	FC0
Gerrit	1/1	FC0
Helios	8/8	FC0
Javaslang	3/3	C3
Jcabi	1/1	C3
Jet	7/14	FC0
Log4J2	25/31	FC0
MapDB	1/1	C3

Figure 9: Subjects with parallel test execution enabled by default.

It came as a surprise the observation that no project used distinct configurations in their modules. Considering our set of 83 projects, we found that only **nine** of those projects had parallelism enabled by default, with only configurations C2, C3, and FC0 being used. Configurations C3 and FC0 were the most popular among these cases. Note that these results under-approximate real usage of parallelism as we used default parameters in our scripts to spawn the Maven process. That decision could prevent execution of particular test modules.

2) Static checking: Given the inherent limitation of dynamic monitoring to find evidence of parallelism, we also looked for indication of parallelism in build files. We parsed all *pom.xml* files under the project's directory and used the same approach as in our previous analysis to classify configurations. We noticed initially that our approach was unable to infer the configuration mode for cases where the decision depends on the input (e.g., `<parallel>${parallel.type}</parallel>`). For these projects, the tester needs to provide additional parameters in the command line to enable parallelization (e.g., `mvn test -Dparallel.type=classesAndMethods`). To handle those case, we considered all possible values for the parameter (in this case, `${parallel.type}`). It is also important to note that this approach is not immune to false negatives, which can occur when *pom.xml* files are encapsulated in jar files or downloaded from the network. Consequently, this static checking is complementary to the dynamic checking, previously presented.

Overall, we found, using this methodology, ten projects that use parallelism. Compared to the set of projects listed in Figure 11, we found two new projects, namely: `Google Cloud DataflowJavaSDK` (using configuration C3) and `Mapstruct` (using configuration FC0). Curiously, we also found that project `Jcabi` was not detected using this methodology. That happened because this project loads its *pom.xml* file from a jar file that we did not check. Considering the static and dynamic methods together, we found a total of 11 distinct projects using parallelism, corresponding to the union of the two subject sets.

> *Answering RQ3: Results indicate that test suite parallelization is underused. Overall, only 13.3% of costly projects (11 out of 83) use parallelism.*

- RQ4. **What are the main reasons that prevent developers from using test suite parallelization?**

To answer this research question we surveyed developers involved in a selection of projects from our benchmark with time-consuming test suites. The goal of the survey is to better comprehend developer's attitude towards the use of parallelism as a mechanism to speedup regression testing. We surveyed developers from a total of 62 projects. From the initial list

[3]This parameter is named `forkMode` in old versions of Maven Surefire

of 83 project, we discarded 11 projects that we knew a priori used parallelization, and 10 projects that we could not find developer's emails from commit logs. From this list of projects, we mined potential participants for our study. More precisely, we searched for developer's name and email from the last 20 commits to the corresponding project repository. Using this approach, we identified a total of 297 eligible participants. Finally, we sent plain-text e-mails, containing the survey, to those developers. In total, 38 developers replied but we discarded three replies with subjective answers. Considering projects covered by the answers, a total of 36 projects (61.29% of the total) were represented in those replies. Note that multiple developers on each project received emails. We sent the following set of questions to developers:

1) How long does it take for tests to run in your environment? Can you briefly define your setup?
2) Do you confirm that your regression test suite does *not* run in parallel?
3) Select a reason for not using parallelization:
   a) I did not know it was possible
   b) I was concerned with concurrency issues
   c) I use a continuous integration server
   d) Some other reason. Please elaborate.

Considering question 1, we confirmed that execution time was compatible with the results we reported in Section IV-A. Furthermore, 12 of the participants indicated the use of Continuous Integration (CI) to run tests, with 4 of these participants reporting that test suites are modularized and those modules are tested independently in CI servers through different parameters. Those participants explained that such practice helps to reduce time to observe test failures, which is the goal of speeding up regression testing. A total of 6 participants answered that they do run tests in their local machines. Note, however, that CI does not preclude low-level parallelization. For example, installations of open-source CI tools (e.g., Jenkins [1]) in dedicated servers would benefit from running tests faster through low-level test suite parallelization.

Considering question 2, the answers we collected indicated, to our surprise, that six of the 36 projects execute tests in parallel. This mismatch is justified by cases where neither of our checks (static or dynamic) could detect presence of parallelism. A closer look at these projects revealed that one of them contained a *pom.xml* file encapsulated in a jar file (similar case as reported in Section IV-B2), in one of the projects the participant considered that distributed CI was a form of parallelism, and in four projects the team preferred to implement parallelization instead of using existing features from the testing framework and the build system — in two projects the team implemented concurrency control with custom JUnit test runners and in two other projects the team implemented concurrency within test methods. Note that, considering these four extra cases (ignored two distributed CI cases), the usage of parallelization increases from 13.3% to 18.1%. We do not consider this change significant enough to modify our conclusion about practical adoption of parallelization (RQ3).

Considering question 3, the distribution of answers was as follows. A total of 8.33% of the 36 developers who answered the survey did not know that parallelism was available in Maven (option "a"), 33.33% of developers mentioned that they did not use parallelism concerned with possible concurrency issues (option "b"), 16.67% of developers mentioned that continuous integration services sufficed to provide timely feedback while running only smoke tests (i.e., short-running tests) locally (option "c"), and 16.67% of developers who provided an alternative answer (option "d") mentioned that using parallelism was not worth the effort of preparing the test suites to take advantage of available processing power. A total of 19.45% of participants did not answer the last question of the survey. The pie chart in Figure 10 summarizes the distribution of answers.

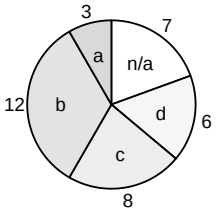


Figure 10: Summary of developer's answers to survey question 3.

Answering RQ4: Results suggest that dealing with concurrency issues (i.e., the extra work to organize test suite to safely explore concurrency) was the principal reason for developers not investing in parallelism. Other reasons included availability of continuous integration services and unfamiliarity with the technology.

C. Speedups

- RQ5. **What are the speedups obtained with parallelization (in projects that actually use it)?**

To answer RQ5, we considered the nine subjects from our benchmark that use parallelization *by default* (see Figure 9). We compared running times with parallelization — configured by project owners — and without parallelization.

Figure 11 summarizes results. Lines are sorted by project names. Columns "*Group*" and "*Name*" indicate, respectively, the group and the name of the subject. Column "T_s" shows sequential execution time and column "T_p" shows parallel execution time. Column "T_s/T_p" shows speedup or slowdown. As usual, a ratio above 1x denotes speedup and a ratio below 1x denotes slowdown.

Results indicate that, on average, parallel execution was 4.4 times faster compared to sequential execution. Three cases worth special attention: `Chaos`, `Jcabi` and `Log4J2`. No significant speedup was observed in `Chaos`, a project with only three test classes, of which one monopolizes the bulk of test execution time. This project uses configuration C2, which runs test classes in parallel and test methods, declared in each class, sequentially. Consequently, speedup cannot be obtained as the cost of the single expensive test class cannot be broken down with the selected configuration. Although project `Jcabi` also uses configuration C2, results obtained are very different compared to `Chaos`. The speedup observed in `Jcabi` was the highest amongst all projects. This project contains

Group	*Subject*	T_s	T_p	T_s/T_p
Medium	Chaos	1.51m	1.47m	1.01x
Medium	Flink	11.79m	2.57m	4.59x
Long	Gerrit	51.19m	40.31m	1.26x
Medium	Helios	4.46m	1.63m	2.73x
Medium	Javaslang	2.18m	1.82m	1.19x
Medium	Jcabi	2.76m	0.30m	9.2x
Medium	Jet	8.26m	3.67m	2.25x
Long	Log4J2	8.24m	8.21m	1.00x
Long	MapDB	10.06m	8.58m	1.17x
average				4.4x

Figure 11: Speedup (or slowdown) of parallel execution (T_p) over sequential execution (T_s). Default parallel configuration of Maven is used. Highest slowdown/speedup appears in gray color.

test classes with a small number of test methods and several methods in those classes are time-consuming. As result, the CPUs available for testing are kept occupied for the most part during test execution. Finally, we note that parallel execution in `Log4J2` was ineffective. We found that Maven invokes several test modules in this project but the test modules that dominate execution time run sequentially by default.

Answering RQ5: Considering the machine setup we used, the average speedup observed with default configurations of parallelization was 4.4x.

- RQ6. **How test execution scales with the number of available CPUs?**

This experiment evaluates the impact of making available to the build system a growing number of CPUs for testing. For that reason, we used a machine with more cores compared to the one described in Section III-A. We used a Xeon E5-2660v2 (2.20GHz) Intel processor machine with 80 virtual CPUs (40 cores with two native threads each) and 256GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). This experiment uses configuration *FC0* as the goal is to evaluate the impact on runtime of spawning a growing number of JVMs in different CPUs. We selected subject MapDB in this experiment as it represents the case of a long-running test suite (see Figure 11) with test cases distributed across many test classes (194 test classes for MapDB). Note that a test class is the smallest unit that can be used to spawn a test job on a JVM and that we have no control over which test classes will be assigned to which JVM that the build system forks.

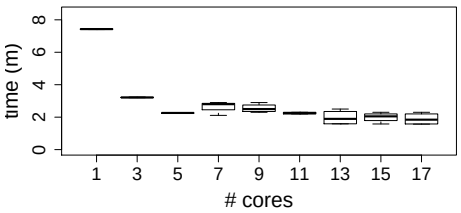


Figure 12: Scalability.

Figure 12 shows the reduction in running times as more CPUs contribute to the execution. We ran this experiment for a growing number of cores 1, 3, ..., 39. The plot shows results up to 17 cores as it is clear the tendency. We noticed that improvements are marginal after three cores, which is the basic setup we used in other experiments. This saturation is justified by the presence of a single test class, `org.mapdb.WALTruncat`, containing 15 test cases that take over two minutes to run.

Answering RQ6: Results suggest that execution FC0 scales with additional cores but there is a bound on the speedup one can get related to how well the test suite is balanced across test classes.

D. Tradeoffs

This dimension assesses the impact of using distinct parallel configurations on test flakiness and speedup. Recall that increased resource contention obtained with parallelism can lead to concurrency issues such as data races that could interfere in results of dependent tests [4], [15]. Flakiness and speedup are contradictory forces that drive configuration selection. Note that Section IV-C evaluated speedup in isolation.

- RQ7. **How parallel execution configurations affect testing costs and flakiness?**

To answer this research question, we selected ten different subjects, ran their test suites against all configurations described in Section II, and compared their running times and rate of test flakiness. We used the sequential execution (configuration *C0*) associated with each project as the comparison baseline in this experiment. We selected subjects from the medium and long groups from Section IV-A and further selected their longest test module for testing. The rationale for this selection was to obtain a diverse set of subjects exposing a wide range of running times (range observed 1.1m-8.2m) and number of test cases (range observed 54-17,513) and yet be able to run the experiment in reasonable time. We ran each project on each configuration with ten seeds. Overall, we needed to reran test suites 600 times, 60 times (10x6 configurations) on each project. Given the low standard deviations observed in our measurements we consider ten seeds reasonable for this experiment.

It is worth mentioning that we used JUnit as opposed to Maven to run configurations *C**. After carefully verifying library versions for compatibility issues and comparing results with JUnit's we observed that several of Maven's executions exposed problems. For example, Maven incorrectly counts the number of test cases executed for some cases where test flakiness are observed. These issues are categorized and documented on our website [2] with Docker containers (for reproduction). To address those issues we implemented basic JUnit test runners (one for each *C** configuration), reflecting the effects of parallel configurations from Maven Surefire. To write those runners we used Maven log files to identify test classes to run and used the Maven dependency plugin [19] to build the project's classpath with the command `mvn dependency:build-classpath`. Once we have the

Subject	*C0*		*C1*		*C2*		*C3*		*FC0*		*FC1*	
	T	*N*	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$
AWS SDK Java (`core`)	3.7m	847	1.95x	2.24%	2.47x	2.77%	3.70x	4.01%	1.85x	0.23%	3.36x	3.1%
Facebook Linkbench	4.3m	98	1.00x	0%	1.65x	1.02%	1.59x	1.02%	1.54x	0%	1.59x	0%
GoogleCloud Dataflow Java (`sdk`)	1.6m	3,345	1.23x	1.67%	2.67x	1.05%	0.80x	5.35%	0.80x	1.70%	0.80x	1.70%
Javaslang (`core`)	1.1m	17,513	1.38x	0%	1.83x	0%	1.38x	0%	1.38x	0%	1.57x	0%
Jcabi Github	2.6m	634	2.10x	0%	17.70x	0%	28.80x	0%	2.00x	0%	2.89x	0%
JCTools (`core`)	3.6m	690	4.50x	0%	3.60x	0%	18.00x	0%	2.77x	0%	9.00x	0%
MapDB	8.2m	5,324	1.52x	0%	2.73x	0%	4.82x	0.05%	1.71x	0.98%	3.42x	0.98%
Moquette	3.7m	169	4.62x	65.64%	3.36x	32.92%	12.33x	77.78%	2.47x	22.53%	9.25x	69.44%
RipMe	1.1m	54	0.94x	0%	1.63x	0%	1.63x	0%	1.37x	0%	1.42x	0%
Stripe Java	4.3m	302	4.78x	6.31%	3.31x	7.31%	21.50x	14.95%	2.69x	0%	8.60x	11.63%
Average	30.3m	2,897.6	2.40x	7.59%	4.10x	4.51%	9.45x	10.32%	1.86x	2.54%	4.19x	8.68%

Figure 13: Speedup versus Flakiness ($\%_{\text{fail}}$). Configuration *C0* denotes the comparison baseline, which runs tests sequentially. Columns T and N indicate time and number of tests, respectively. Other columns show speedup and percentage of failing tests in different configurations, compared to *C0*.

classpath and the tests to run, we invoke our JUnit runner that runs the test classes with different parallel configurations at once. For the sequential execution, we invoke the default JUnit test runner. For parallelism within a JVM (i.e., the configurations C1, C2, and C3), we configured the JUnit runner with ParallelComputer [12] that internally uses a pool of threads with cache to run test classes and methods. For the configurations with forked JVMs (i.e., FC0 and FC1), we have a queue of test classes and a pool of three threads that "consumes" the next test class and spawns a JVM to run it. Recall that in our setup (see Section III-A), we limited our kernel to use only three cores and reserved one for OS-related processes. To verify that our approach satisfies the purpose of this experiment, we executed some subjects on Maven and compared the performance with our artifacts and observed insignificant differences (a fraction of seconds in most cases). To assure that our experiments terminate (recall that deadlock or livelock could occur) we used the `timeout` command [16] configured to dispatch a *kill* signal if test execution exceeds a given time limit. Finally, we saved each execution log and stack traces generated from JUnit to collect the execution time, the number of failing tests, and for reference to analyze and diagnose outliers in our results.

Figure 13 summarizes results ordered by subject's name. Recall that we only executed one module for each subject. Values are averaged across multiple executions. We did not report standard deviations as they are very small in all cases. Considering flakiness, results show that 0% of flakiness have been reported in 26 of the 60 combinations analyzed (43% of the total) and that for seven of the ten projects 0% flakiness was reported in at least one of the configurations. The projects with flakiness in all configurations were `AWS SDK`, `GoogleCloud`, and `Moquette`. It is worth highlighting the unfortunate case of `Moquette`, which manifested more than 20% flaky tests in every configuration. Considering time, it is noticeable from the averages, perhaps as expected, an increasing speedup from configuration *C1* to *C3* and from configuration *FC0* to *FC1*. It is also worth mentioning that some combinations manifested slowdown instead of speedup. Recall that parallel execution introduces the overhead of spawning and managing JVMs and threads.

> *Answering RQ7: Overall results indicate that the test suites of 70% of the projects we analyzed could be run in parallel without manifesting any flaky tests. In some of these cases, speedups were significant, ranging from 1x to 28.8x.*

V. Discussion

This paper reports our finding on a study to evaluate impact and usage of test suite parallelization, enabled by modern build systems and testing frameworks. This study is important given the importance to speedup testing. Note that test suite parallelization is complementary to alternative approach to speedup testing (see Section VII). The observations we made in this study trigger multiple actions:

- *Incentivize forking.* Forked JVMs manifest low rates of test flakiness. For instance, in *FC0*, only 4 of 10 projects manifest flakiness and, excluding the extreme case of `Moquette`, projects manifest flaky tests in low rates 0.23% to 1.70%. Developers of projects with long-running test suites should consider using that feature, which is available in modern build systems today (e.g., Maven).
- *Break test dependencies.* Non-forked JVMs can achieve impressive speedups at the expense of sometimes impressive rates of flakiness. Breaking test dependencies to avoid flakiness and take full advantage of those options is advised for developers with greater interest in efficiency.
- *Refactor tests for load balancing.* Forked JVMs scales better with the number of cores when the test workload is balanced across testing classes. To balance the workload, automated user-oblivious refactoring can help in scenarios where developers are not willing to change test code but have access to machines with a high number of cores.
- *Improve debugging for build systems.* While preparing our experiments, we found scenarios where Maven's executions did not reflect corresponding JUnit's executions. (Docker

reproduction scripts available.) Those issues can hinder developers from using parallel testing. Better debugging infrastructure is important.

This study brings to light the benefits and burdens of test suite parallelization to improve test efficiency. It provides recommendations to practitioners and developers of new techniques and tools aiming to speed up test execution with parallelization.

VI. Threats to Validity

The main threats to validity of this study are the following. *External Validity:* Generalization of our findings is limited to our selection of subjects, testing framework, and build system. To mitigate that issue, we selected subjects according to an objective criteria, described in Section III. It remains to evaluate the extent to which our observations would change when using different testing frameworks and build systems. *Internal Validity:* Our results could be influenced by unintentional mistakes made by humans who interpreted survey data and implemented scripts and code to collect and analyze data. For example, we developed JUnit runners to reproduce Maven's parallel configurations. All those tasks could introduce bias. To mitigate those threats, we worked in pairs as to increase chances of capturing unintentional mistakes. *Construct Validity:* We considered a number of metrics in this study that could influence some of our interpretations. For example, we measured number of test cases per suite, distribution of test costs in a suite, time to run a suite, etc. In principle, these metrics may not reflect the main problems associated with test efficiency.

VII. Related Work

Researchers and practitioners have been shedding light to the to the demand of techniques for optimizing test execution. For instance, a recent paper from the automotive industry reported that test suites can take days to run [3]. There are different aspects to consider when dealing with the challenge of reducing test cost. Research has focused mostly on test suite minimization, prioritization, reduction, and selection [27]. Most of these techniques are unsound (i.e., they do not guarantee that fault-revealing tests will be selected), however, more recently, sound techniques have been proposed [8], [24]. For example, to soundly select tests for execution, Ekstazi [8], [9] conservatively computes which tests have been impacted by changes. A test is discarded for execution if it does not depend on any changed file dynamically reachable from execution. Although Ekstazi is focused on regression test selection, it highlights the importance of parallelism as a complement to test selection. In fact, as in their evaluation, we also discovered subjects with parallelism enabled by default. Test suite parallelization is complementary to regression testing techniques.

[[Please: (i) consider categorizing related work in sections (ii) list papers that need to be discussed (iii) discuss papers that use multi-execution (SIMD cpus like GPUs) to speedup testing **]]**

VIII. Conclusions

Testing is expensive. Despite all advances in regression testing research, dealing with high testing costs remains an important problem in Software Engineering. This paper reports our findings on the usage and impact of test execution parallelization in open-source projects. Multicore CPUs, as well as testing frameworks and build systems to capitalize on them, are widely available today. Despite some resistance observed from practicioners, our results suggest that parallelization can be done in many cases without sacrificing reliability and more research needs to be done to improve automation (e.g., breaking test dependencies and refactoring test suites) as to safely optimize parallel execution. The artifacts we produced as result of this study are available from the following web page https://doubleblind.000webhostapp.com/.

References

[1] Jenkins ci. https://jenkins.io/, 2017.

[2] Our web page. https://doubleblind.000webhostapp.com/, 2017.

[3] S. Arlt, T. Morciniec, A. Podelski, and S. Wagner. If a fails, can b still succeed? inferring dependencies between test results in automotive system testing. In *ICST*, pages 1–10, April 2015.

[4] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *ESEC/FSE*, pages 770–781, 2015.

[5] Unix StackExchange community. Using `isolcpus`. http://unix.stackexchange.com/questions/326579/how-to-ensure-exclusive-cpu-availability-for-a-running-process, 2017.

[6] Github. Github about stars, 2017. https://help.github.com/articles/about-stars/.

[7] Github. Github api web site, 2017. http://developer.github.com/v3/search/.

[8] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.

[9] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection (website), 2017. http://ekstazi.org/research.html.

[10] GoogleTechTalks. Tools for continuous integration at google, October 2010. http://www.youtube.com/watch?v=b52aXZ2yi08.

[11] JUnit. Junit web site, 2017. http://junit.org.

[12] JUnit. Parallelcomputer (junit api), 2017. http://junit.org/junit4/javadoc/4.12/org/junit/experimental/ParallelComputer.html.

[13] Kernel.org. Kernel linux options. https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html?highlight=isolcpu, 2017.

[14] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*, pages 257–267, 2013.

[15] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[16] Linux man Page. timeout - run a command with a time limit. http://linux.die.net/man/1/timeout, 2017.

[17] Maven, 2017. https://maven.apache.org/.

[18] NUnit. Nunit web site, 2017. http://www.nunit.org.

[19] Maven Dependency Plugin, 2017. https://maven.apache.org/plugins/maven-dependency-plugin/.

[20] Maven Surefire Plugin, 2017. http://maven.apache.org/surefire/maven-surefire-plugin/.

[21] Chandra Prasad and Wolfram Schulte. Taking control of your engineering tools. *Computer*, 46(11):63–66, 2013.

[22] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[23] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 281–, Washington, DC, USA, 2003. IEEE Computer Society.

[24] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. Change-based test selection: An empirical evaluation. *Empirical Softw. Engg.*, 21(5):1990–2032, October 2016.
[25] TestNG. Testng web site, 2017. http://testng.org.
[26] Google Engineering Tools. Testing at the speed and scale of google, June 2011. http://google-engtools.blogspot.com.br/2011/06/testing-at-speed-and-scale-of-google.html.
[27] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability*, 22(2):67–120, March 2012.