Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem

Alex Gyori, Owolabi Legunsen, Farah Hariri, Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801, USA
{gyori,legunse2,hariri2,marinov}@illinois.edu

Abstract—**Regression testing in very large software ecosystems is notoriously costly, requiring computational resources that even large corporations struggle to cope with. Very large ecosystems contain thousands of rapidly evolving, interconnected projects where *client* projects transitively depend on *library* projects. Regression test selection (RTS) reduces regression testing costs by rerunning only tests whose pass/fail behavior may flip after code changes. For single projects, researchers showed that *class*-level RTS is more effective than lower method- or statement-level RTS. Meanwhile, several very large ecosystems in industry, e.g., at Facebook, Google, and Microsoft, perform *project*-level RTS, rerunning tests in a changed library and in all its transitive clients. However, there was no previous study of the comparative benefits of class-level and project-level RTS in such ecosystems.**

We evaluate RTS *opportunities* in the MAVEN Central open-source ecosystem. There, some popular libraries have up to 924589 clients; in turn, clients can depend on up to 11190 libraries. We sampled 408 popular projects and found that 202 (almost half) cannot update to latest library versions without breaking compilation or tests. If developers want to detect these breakages earlier, they need to run very many tests. We compared four variants of class-level RTS with project-level RTS in MAVEN Central. The results showed that class-level RTS may be an order of magnitude less costly than project-level RTS in very large ecosystems. Specifically, various class-level RTS variants select, on average, 7.8%–17.4% of tests selected by project-level RTS.

I. INTRODUCTION

Very large software ecosystems are becoming more common in both industry and in the open-source community. A very large ecosystem contains thousands of rapidly evolving, interconnected projects where *client* projects transitively depend on *library* projects. Examples of such ecosystems in industry include those at Facebook, Google, and Microsoft, where many software projects typically reside in a monolithic repository and clients are built and tested against latest library versions whenever the library changes. Open-source ecosystems also exist for popular programming languages—e.g., MAVEN Central for Java [1], NPM for JavaScript [2], NUGET for .Net [3], CRAN for R [4], RUBYGEMS for Ruby [5], PYPI for Python [6], etc.—even if the source code is not centralized in one monolithic repository.

Regression testing [7], [8], [9], [10], [11], [12], [13], [14], [15] is notoriously expensive [16], [17] in very large software ecosystems but still valuable to quickly detect whether library changes break some clients. Several large proprietary software organizations have developed regression testing systems, e.g., Facebook's Buck [18], Google's TAP [16], [19], and Microsoft's CloudBuild [17]. When a library changes, the tests in the library, and, ideally, the tests in all its clients, are rerun to check for regressions. Even in open-source ecosystems, which lack the centralized governance and regression testing systems that these companies have, regression testing of client code after library changes can be highly beneficial: (i) clients' tests can test library code in ways that library developers did not foresee, (ii) library developers can more quickly see if their changes break some popular clients, (iii) client developers can more quickly discover how library changes break their code (even if they choose not to upgrade to the latest library version), and (iv) client developers might find it easier to incrementally update their code as libraries evolve.

Regression test selection (RTS) [13], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] aims to reduce the cost of regression testing by rerunning only tests whose pass/fail behavior may flip due to the code changes. An RTS technique works by tracking test dependencies on an old code version. Then, only tests for which some dependency changed are rerun in the new code version. RTS techniques vary in the granularity at which they select tests and compute test dependencies, e.g., at statement, block, method, class, or project levels. Tracking test dependencies can be done statically or dynamically.

There is a gap between the research on RTS, which so far mostly evaluated individual projects, and RTS practiced in several very large software ecosystems in industry, e.g., at Facebook, Google, and Microsoft. Specifically, researchers showed that, for individual projects, *class*-level RTS can be more beneficial than RTS only at lower granularities, e.g., method-level [28], [29], [30]. Meanwhile, regression testing systems for very large ecosystems (e.g., Buck, CloudBuild, and TAP) perform *project*-level RTS. Yet, no prior study compared class-level and project-level RTS in very large software ecosystems. Section II shows via examples some benefits that class-level RTS can provide over project-level RTS. While these benefits would be ideally evaluated in industry, we cannot easily access the proprietary systems and code.

In this paper, we evaluate RTS opportunities by comparing class-level with project-level RTS in the MAVEN Central open-source ecosystem. There, popular libraries have up to 924589 clients; in turn, clients depend on up to 11190 libraries. This underapproximates the number of clients because we only count clients deployed in MAVEN Central. To show that

MAVEN Central libraries evolve in a way that breaks clients, we first conducted a small formative study on a sample of 408 popular GitHub Java projects released in MAVEN Central for which we could successfully run tests. 202 projects (i.e., almost half) could not update to the latest versions of their libraries because some libraries broke the clients' compilation, some broke tests, and some broke both. While not all libraries care about breaking all clients and not all clients want to update to the latest library versions, developers who do care about detecting such breakages earlier could benefit from a regression testing system in the MAVEN Central ecosystem. Some Apache projects started an effort on Gump [31], but MAVEN Central has no widely used regression testing system.

Our comparison of class-level and project-level RTS in MAVEN Central showed that *class-level RTS can be an order of magnitude cheaper than project-level RTS* in very large ecosystems. We compared four variants of class-level RTS with project-level RTS; MAVEN Central projects are Jar files, so we use "Jar" and "project" interchangeably. We refer to the project-level RTS as JJ: it computes both dependencies and changes at the **J**ar granularity. When a library changes, JJ reruns *all tests in the library and all tests in all the library's transitive clients*. JJ mimics what companies with an abundance of resources do for RTS, but even these companies have reported the increasing costs of JJ [16], [19], [32]. The question we ask is how much could be saved by lower-granularity, class-level RTS.

We compared project-level RTS against four class-level RTS variants that all track dependencies at the class level but differ in whether they track changes at the class or Jar level, and whether they compute dependencies statically or dynamically. We used Java projects that release Jars in MAVEN Central and keep source code in GitHub. We performed the comparison on 13961 change sets in 168 libraries that have test Jars and a total of 580876 clients. The results for this very large software ecosystem showed that class-level can be much cheaper than project-level. Specifically, various class-level RTS variants select, on average, only 7.8%–17.4% of the tests that project-level RTS selects.

This paper makes the following contributions:

⋆ **Comparison of RTS Techniques at Scale:** We are the first to empirically compare RTS techniques at class and project levels of granularity of dependencies and changes for both static and dynamic RTS techniques at scale.

⋆ **Empirical Evaluation:** Our study is the largest study so far of RTS for open-source projects. We compare five RTS techniques using 13961 change sets in 168 projects while performing RTS in 580876 clients in MAVEN Central.

⋆ **Formative Study:** We show that client-library breakages occur in the MAVEN Central ecosystem. Almost half of 408 clients in our formative study cannot safely update to their libraries' latest versions, and 41.3% of breakages manifested as test failures. Yet, library developers have no good way to test whether their changes break clients.

The scripts we used for this work are available online at http://mir.cs.illinois.edu/issre2018.zip

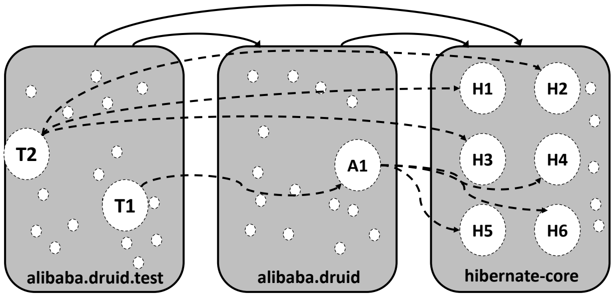

Fig. 1. Two test dependency levels for `alibaba.druid`: Jar shown by solid edges and Class shown by dashed edges

II. MOTIVATING EXAMPLES

We provide three examples to (i) illustrate the benefits that class-level can provide over project-level RTS, (ii) motivate the need for better regression testing in very large open-source ecosystems, and (iii) show that regression testing is critical in very large ecosystems, even for seemingly harmless changes.

A. Potential Benefits of Class-Level over Project-Level RTS

Consider `alibaba.druid` [33], which is is an important library released in MAVEN Central [34]—it has 52 *direct* clients in MAVEN Central. `alibaba.druid` is also a client of many libraries, depending *directly* on 46 libraries. At the time of our study, `alibaba.druid` depends on `hibernate-core` version 5.1.0 but the latest version of `hibernate-core` is 5.2.4. All `alibaba.druid` tests pass with `hibernate-core` 5.1.0 but one test (`HibernateCRUDTest`) fails with `hibernate-core` 5.2.4. The `alibaba.druid` and `hibernate-core` developers were likely unaware of this failure *right after* the change broke the test. If `alibaba.druid` developers decide to update `hibernate-core`, they will have to handle this issue, at which point it is too late for `hibernate-core` developers to reconsider their change from 5.1.0 to 5.2.4. Had the developers been immediately informed about the impact of the library change on the client, they may have made different decisions. Regression testing can help to detect such breakages earlier.

Figure 1 shows both project-level and class-level test dependencies from `alibaba.druid` to `hibernate-core`. Solid arrows show project-level test dependencies and dashed arrows show class-level dependencies. At the project level, if `hibernate-core` changes, *all* 1912 test classes in `alibaba.druid` will be rerun. In contrast, only *two* test classes in `alibaba.druid` can reach `hibernate-core`; `HibernateCRUDTest` (shown as `T2`) directly accesses three `hibernate-core` classes (`H1`, `H2`, `H3`), while `DruidConnectionProviderTest` (`T1`) uses `DruidConnectionProvider` (`A1`) which implements three `hibernate-core` interfaces (`H4`, `H5`, `H6`). *At most two tests* (`T1` and `T2`) could be selected through Class dependencies for any change to `hibernate-core`. In fact, we ran class-level

```
public static Setter create(Field f, Object bean) {
+  if (Modifier.isFinal(f.getModifiers()))
+    throw new IllegalStateException(...);
   if(f.gettype().isarray())
     return new arrayfieldsetter(bean,f);
```

Fig. 2. Code change in commit 6e11f89d in ARGS4J

RTS on 107 commits and found that *usually only one* test (T2) gets selected (three hibernate-core interfaces were stable and did not change, so T1 did not get selected). For some commits, *zero tests* were selected (changes in hibernate-core were not reachable from classes H1–H6). This example shows that class-level RTS can substantially reduce the number of tests run, compared with project-level RTS.

B. Need for Regression Testing in Open-Source Ecosystems

Guava is a popular open-source library for creating and manipulating several collection types. Guava is maintained by Google and is partially mirrored publicly on GitHub. Guava's GitHub commit 73e382 [35] modifies 170 files; this commit accumulates several internal commits that are exported as a single public commit. Accumulated commits are common in some open-source repositories that mirror proprietary projects.

Google runs the TAP regression-testing system, which tests all Guava changes against all its clients that are *internal* to Google but does not test Guava against clients *external* to Google. Commit 73e382 breaks Square OkHttp, an open-source client that depends on Guava. While Google developers may have expected this commit to break some clients, they likely did not know *how many* external clients would break. Moreover, they likely did not know that some client(s) will break *right after* they changed a subset of the 170 files in a small internal commit. Google could afford to run some important external clients on the TAP system. However, the open-source ecosystem has no regression testing system like TAP to quicker detect library changes that break some clients. Our formative study (Section III) shows that breakages do occur in the MAVEN Central open-source ecosystem, so developers may want to run regression testing across projects.

C. All Changes Should be Regression Tested in the Ecosystem

Figure 2 shows an example change in ARGS4J [36]. ARGS4J is a fairly popular library; its artifacts in MAVEN Central have 17006 direct and indirect clients (and potentially many more clients that are not deployed to MAVEN Central). The change in Figure 2 modifies the Setter object to throw an IllegalStateException if a Setter is created for a final field. Before this change, a different exception was thrown.

Adding a new exception seems innocuous, but it breaks at least one ARGS4J client, Google Closure Templates, a project in our formative study (Section III). We first found that updating the ARGS4J version from 2.0.23, the version declared in the Google Closure Templates build file, to 2.33 breaks a Google Closure Templates test. We then used git bisect to find the specific commit between these two ARGS4J versions that breaks the Google Closure Templates test. Had ARGS4J developers been aware of this issue, they may have reconsidered making the change, or they may have informed the Google Closure Templates developers to update their code.

D. Improving Regression Testing in Very Large Ecosystems

In an open-source ecosystem, developers may choose to run tests from multiple projects after code changes, e.g., they may run the tests of a library that changed and the tests of some/all of its clients. The goal is to test that the library changes do not break the clients, which is hopefully reflected as test failure(s) in the client tests. The sooner library developers discover test failure(s) in the clients' tests, the better they may decide how to proceed. We envision that library developers would select clients that are relevant to run (e.g., they may choose to run only tests in "important" clients), or client developers may choose to run their own tests whenever one of their libraries changes. It is up to developers to decide how to handle information on failing tests. Library developers may decide to create a patch script to apply on all the breaking clients when possible; alternatively, they may decide to revert changes to avoid breaking (important) clients or may inform clients that changes break some tests. Client developers may refine their code to handle library changes. Facebook's Buck, Google's TAP, and Microsoft's CloudBuild users already get such timely information. Open-source developers should also get similar information. This paper evaluates RTS techniques that can provide such information *faster*, improving regression testing in open-source ecosystems as well as in proprietary systems like Buck, CloudBuild, and TAP.

III. FORMATIVE STUDY

The goal of our formative study is to quantify how often library updates break clients' compilation or tests in MAVEN Central. If breakages do not happen, there would be no need to test clients when libraries change. If breakages do happen, they can be detected quicker with RTS run for the entire ecosystem. We *do not assume that all libraries care about all clients or that all clients need to always update to the latest library versions*. In fact, researchers already showed that library developers make backwards-incompatible changes [10], [37], [38] and that clients do not always update [37], [39], [40]. However, if libraries and clients care about breakages, obtaining more precise information about such breakages more frequently and cheaply can be highly beneficial for decision making by both library and client developers. Knowing the precise number of (important) clients that a library change breaks can help to better decide whether to proceed with the change. Dually, client developers may benefit from knowing whether each library change breaks their code to decide whether to incrementally co-evolve their code (if breakages occur), or ignore the update (if no breakages occur).

For our study, we used GitHub Java projects that use MAVEN, the most popular build system for Java. Many prior projects [37], [38], [39], [40], [41], [42], [43], [44], [45], [46] studied the impact of library updates on clients, e.g., stale dependencies, but we are the first to perform such a study with a view to evaluate RTS opportunities in a very large software ecosystem. Section VIII discusses specific differences between our formative study and these related studies. In brief, we consider *both* compilation and test failures of clients, and not just statically computed metrics [10], [47], [48], [49].

Our formative study analyzes the staleness in Maven Central and the breakages it causes from three perspectives of (i) dependencies as pairs of client and library, (ii) clients, and (iii) libraries. We answer the following research questions: **RQ1 (Dependency View):** How common are stale dependencies, i.e., a newer version of the library is available, among Maven-based projects? **RQ1.1** How often would clients break if updated? **RQ1.1.1** How often would updates break compilation? **RQ1.1.2** How often would updates break tests? **RQ2 (Client View):** How many projects use stale dependencies, i.e., do not use the latest version of a library available? **RQ2.1:** How often can clients not be updated to latest versions of their libraries? **RQ2.2:** How often do updates break client compilation? **RQ2.3:** How often do updates break client tests? **RQ3 (Library View):** How many unique libraries cannot be updated to the latest version? **RQ3.1:** How many unique libraries break client compilation? **RQ3.2:** How many unique libraries break client tests?

Our formative study considers 408 projects selected from among the most popular 3000 GitHub Java projects. All 3000 projects contain a `pom.xml` file in their root directories, indicating that they likely use Maven. The 408 projects in our corpus are those that remain after filtering out projects that we could not build, or for which some tests fail on the latest version at the time of our study. We excluded projects with failing tests as those may indicate instability or misconfiguration on our end. These 408 projects have a total of 12231 direct dependencies. We ran all formative study experiments on an Ubuntu 16.04 VM with Oracle Java 8_u121.

For each project in our corpus, we first check whether the project has any *stale* dependency, i.e., whether the project depends on a library that has some newer version released in Maven Central. We use Maven's `versions` plugin to check for stale dependencies. Specifically, the command `mvn versions:display-dependency-updates` finds all (direct) dependencies with newer versions released in Maven Central. Additionally, the command `mvn versions:display-property-updates` handles the case where projects use build properties (similar to variables) to declare dependency versions.

Table I summarizes the results of our formative study. The number of stale dependencies (6828) is *larger* than the number of non-stale dependencies (5403), which answers **RQ1:** 6828 (55.8%) dependencies, pairs of client and library, are stale, i.e., a library has a newer version. It also answers **RQ2:** 375 (91.9%) of 408 projects have at least one stale dependency. The high percentage shows that even popular projects do not always update their libraries to the latest version.

For each project with some stale dependency, we first update *at once* all the dependencies to their latest possible versions, to mimic a global update of all dependencies. After the global update, we recompile and run all the tests for the project. If compilation or tests fail, then some library update broke the client. We get the answer for **RQ2.1:** out of 375 projects with at least one stale dependency, 202 (53.9%) cannot trivially update all their dependencies to the latest version because either compilation or tests break. The fact that more than half of projects with stale dependencies cannot update their dependencies shows that updating is non-trivial. (Note: 202 may underestimate projects that we could not update; even if a project compiles and passes tests with the update, it may still break for scenarios not covered by tests.)

TABLE I
Formative Study Results

RQ	Statistic	Abs	Relative(%)
	Non-Stale Dependencies	5403	44.2%
RQ1	Stale Dependencies	6828	55.8%
RQ1.1	Breaking Updates	812	11.8%
RQ1.1.1	Compilation Breaking Updates	476	58.6%
RQ1.1.2	Test Breaking Updates	336	41.3%
	Projects in Corpus	408	*n.a.*
RQ2	Projects with Stale Dependencies	375	91.9%
RQ2.1	Non-updateable Projects	202	53.9%
RQ2.2	Compilation-Breaking Projects	176	46.9%
RQ2.3	Tests-Breaking Projects	101	26.9%
RQ3	Unique Libraries	2133	*n.a.*
RQ3.1	Compilation-Breaking Libraries	320	15.0%
RQ3.2	Tests-Breaking Libraries	239	11.2%

For 202 projects that failed our global updates, we roll back the global updates, update each dependency *one by one*, and run the tests. We find that several dependencies can independently break each project's compilation or tests. **RQ1.1:** 812 (11.8%) of the stale dependencies break the projects when updated; **RQ1.1.1:** 476 (58.6%) of dependencies cause compilation failure; **RQ1.1.2:** 336 (41.3%) of dependencies cause test failures in their clients. While 11.8% is a relatively low percentage of stale dependencies that break the projects, at least one such dependency affects more than half the projects, 53.9%. Breakdowns of compilation and test failures per project also have high percentages; **RQ2.2:** 176 (46.9%) of projects with stale dependencies have at least one dependency that breaks the project compilation when updated; **RQ2.3:** 101 (26.9%) of these projects have at least one dependency that breaks at least one project's test when updated.

Finally, we analyzed breakages from the view of libraries. After all, even if a large number of dependencies break, and a large number of projects are affected, all the issues could, in theory, stem from just a tiny number of widely used libraries. For **RQ3:** our corpus has 2133 unique libraries. In Maven, libraries are uniquely identified by *GAV* triples: *group*, *artifact*, and *version*. For **RQ3**, we count a library as unique based on the pair of group and artifact. We found that a significant percentage of libraries break their clients when updated to the latest version; **RQ3.1:** 320 (15.0%) of the libraries break compilation in at least one client; **RQ3.2:** 239 (11.2%) of the libraries break at least one test in at least one client.

In sum, we quantified the breakages that occur for library updates in very large ecosystems like Maven Central. Importantly, 41.3% of breakages manifested as test failures in clients, offering opportunities for regression testing to help quicker detect such cases for library-client pairs that care about breakages. The updates that break tests could be detected even quicker and cheaper using RTS rather than running all tests.

Semantic Versioning. One concern is whether the breakages we identify are intentional or not, and whether library developers are already aware that many of their changes are backwards-incompatible. One approach to answer this question would be to ask developers, but this approach would be hard to scale in practice. An approach that developers use to signal when a change in a library is intended to break backwards compatibility is semantic versioning [50]. Projects that follow semantic versioning indicate backwards-incompatible changes by changing the major version number; new features and other backwards-compatible changes would only modify the minor version number, while small changes (which still maintain backwards compatibility) would change only the patch number.

While our study did not aim to evaluate whether projects use semantic versioning or not, our findings generally confirm the findings of Raemaekers et al. [51] that many projects do not follow semantic versioning. For example, `hibernate-core`'s backwards-incompatible change that broke `alibaba.druid` was only a minor version change form 5.1.0 to 5.2.4. Similarly, the change in ARGS4J that broke Google Closure Templates was also a minor change from version 2.0.23 to 2.33. While the fact that developers introduced backwards incompatibility in changes to only the minor version does not unequivocally show they were not aware that the changes are backwards-incompatible, it illustrates a gap in even being able to identify whether a change is or is not backwards compatible. Ecosystem-level testing of clients could even assist developers into following semantic versioning more rigorously.

IV. Choosing Libraries, Clients, and Changes

This section describes how we set up our experiments to compare class- and project-level RTS in the MAVEN Central open-source ecosystem. An important objective of our evaluation is to identify a *large* number of libraries, clients, code changes, and tests so that the results are more representative of very large software ecosystems. Therefore we needed to (1) identify popular libraries in MAVEN Central for which we can map release versions to GitHub changes, (2) identify clients for these libraries, and (3) evaluate how many library and client tests would have been selected by each technique after real code changes. Section IV-A describes how we identified libraries. Section IV-B describes how we identified clients. Section IV-C describes how we performed RTS.

A. Finding Popular Libraries with GitHub Changes

Mapping from Libraries to Source code is Challenging. For our experiments, we need to identify libraries that have (1) code changes on GitHub, and (2) many clients with tests. Meeting both requirements is challenging because the data for each is in different stores of information, and we need to map the information from these stores. The best place to identify libraries with code changes is in open-source repositories like GitHub [52] but such repositories do not store data in ways that make it easy to identify the libraries' clients. One can easily find the libraries on which a client depends (e.g., `"mvn dependency:list"` lists all dependencies—Jar files that a Maven-based project transitively depends on), but one cannot easily find all clients of a library. More so, mapping dependencies which are listed as Jar files to GitHub repositories is not trivial, and mapping to specific commits that produced the Jar files is even harder. On the other hand, MAVEN Central is a great place to identify clients for libraries but it does not map Jars to source code repositories. Due to these challenges, we had to map the information stored between MAVEN Central and GitHub.

Mapping from GitHub to MAVEN Central is Better. When identifying evolving libraries for use in our experiments, we mapped from GitHub data about the libraries to MAVEN Central data. In theory, one can map data from either repository to the other in either direction. In practice, however, we found three reasons why identifying the source-code repository for a randomly selected MAVEN Central Jar can be less optimal, even when the `pom.xml` file contains a URL for the repository. First, the Jar may be old and its repository may not exist because the project was discontinued, is private, or was moved to another repository altogether. Second, even if a repository is found, it could be using a different or older version-control system (e.g., SVN or CVS). Third, an old Jar can mean that code changes for that project are not representative of modern software development best practices. Researchers have studied how commit patterns differ between centralized and distributed version-control systems [53].

Process for Selecting Evolving Libraries. Given the aforementioned challenges, we selected libraries for our experiments using the following process: (i) select an initial set of projects based on GitHub popularity, (ii) map GitHub repositories to MAVEN Central Jar files that were likely built from these repositories, and (iii) map the GitHub commits to the MAVEN Central Jar versions.

(i) Selecting an Initial Set of GitHub Projects. We started from the top 3000 Java projects on GitHub (ranked by the number of stars and forks) which contain a `pom.xml` file in the root directory. We chose Maven because it is still the most popular build system for Java, many projects that use Maven release their Jars on MAVEN Central, and our tool-chain was developed to work with Maven.

(ii) Mapping GitHub Repositories to MAVEN Central Jars. It is extremely tedious to read through thousands of `pom.xml` files from GitHub to find which of them produced each Jar. We automated this by attempting to build Jars from each GitHub repository, using the command, `mvn install -DskipTests`. The build command succeeded for only 1901 of the 3000 top projects. Out of these, a few projects do not create Jars, and some projects created Jars but did not put them in the local cache where most MAVEN projects typically install their Jars. For projects that installed successfully to the Maven cache, we relied on the default Maven naming convention to map the created Jars to Maven Central Jars. Maven uniquely identifies Jars by the GAV triple g:a:v, short for $g$roupId:$a$rtifactId:$v$ersion. The install command names Jars as a-v.jar. We ignored the version in our mapping, because latest commit in project's GitHub repository typically produces version of the Jar that does not match any of the versions released on Maven Central. However, the code version for older GAVs is still in the same repository. We successfully mapped 1204 GitHub projects to 7954 Maven Central Jars. Note that one project can create multiple Jars; the number of Jars mapped per project ranged from 1 to 232.

(iii) Mapping GitHub Commits to Maven Central Jar Versions. Maven's default automated release workflow creates a Git tag using the corresponding commit id. For each GitHub repository that we successfully mapped to a Maven Central Jar, we searched through the repository's Git tags for tags that match a release versions, resulting in a mapping from commit to Maven Central GAV. For each repository, we analyze changes for only one Jar version so as not to bias our results. We picked the most popular version of the Jar, i.e., version with the most clients in Maven Central (Section IV-B describes client identification). We select up to 100 commits created after the selected release tag but before the next release tag. In sum, we used actual changes that developers made to the code for the Jars released in Maven Central, and we used changes only between two tags/versions, so even if projects follow semantic versioning, all commits we selected are for the same version.

B. *Identifying Clients for Selected Libraries*

Process for Identifying Clients. For each library that we selected in Section IV-A, we computed the transitive closure of all GAVs in the Maven Central graph that depend on at least one node that corresponds to the library. We then found how many of these GAVs release test classes. One GAV can release several Jars, e.g., one Jar containing application binaries, one Jar containing sources, and one Jar containing binaries for test classes. A convention on Maven Central is that a-v-tests.jar contains test classes for the binaries in a-v.jar. Out of 1204 libraries from Section IV-A, only 215 have corresponding test Jars on Maven Central. (Section VII discusses threats to validity, including that many project do not release their tests on Maven Central.) We further excluded 13 projects for which we found no code changes corresponding to Maven Central Jars, and 34 projects which had more tests than all their clients combined.

Ratios of Client to Library Jars. We analyzed 168 libraries, 46 of which released their own test Jars on Maven Central. The number of transitive clients for these libraries ranges from 1 to 443306 (average 22884.6). The number of transitive clients with tests ranges from 1 to 75923 (average 3593.3). Figure 3 shows the number of client and test Jars for the libraries that we use. Seven libraries have more test Jars than client Jars that depend on them, because some GAVs release only test Jars, and some release web-archives (.war or .ear) which we exclude. Figure 3 shows that not all client Jars have corresponding test Jars (difference of "Client Jars" and "Tests Jars" lines), many such client Jars contain tests.

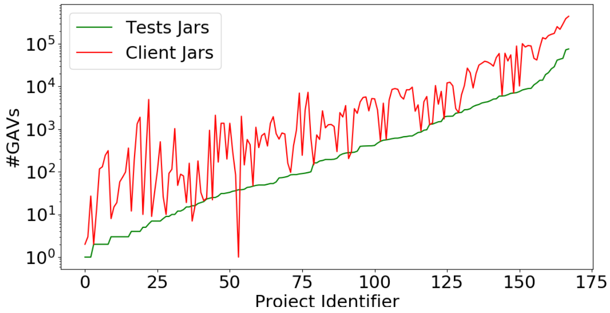

Fig. 3. Total clients and clients with tests for projects under analysis

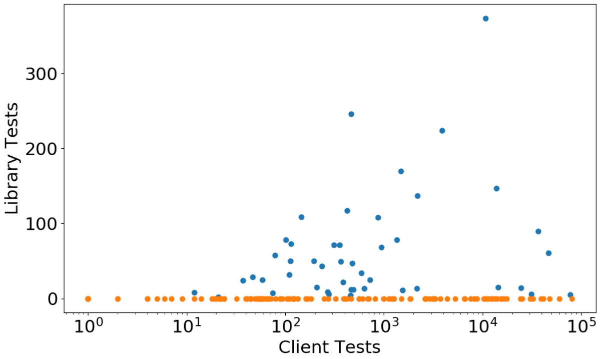

Fig. 4. Tests in client vs. tests in library itself

Ratios of Client to Library Tests. Figure 4 plots the number of client tests to library tests. Each dot in the plot corresponds to a library; libraries with 0 tests in the Jar are depicted with a different color. Figure 4 also shows that libraries contain relatively much fewer tests than their clients. The number of test classes in libraries ranges from 0.0 to 374.0 (average 17.0). The number of test classes in clients ranges from 1.0 to 80460.0 (average 5462.4). Most importantly, for libraries that contain their own tests, the ratio of client's test classes (excluding tests in the library) to test classes in the library ranges from 1.3 to 15441.6 (average 555.3). This ratio represents how many more tests library developers have to run in addition to their own tests, if they run all clients' tests. The goal of RTS is to reduce the cost of running these tests.

C. *Evaluated Dimensions of RTS*

We describe different RTS aspects that we explored and our experimental setup for comparing class-level and project-level RTS. The inputs to RTS are two program versions together with their tests. RTS computes changes between the two versions and intersects them with the test dependencies in the old version, i.e., program elements that each test depends on. Then RTS reruns in the new version only tests for which some dependency changed, i.e., *affected* tests. In our experiments, we explored alternatives along three RTS aspects: (i) change computation, (ii) test dependency computation, and (iii) analysis of test dependencies to find affected tests.

Granularity of Code Changes and Test Dependencies. Changes and test dependencies can be computed at different granularity, e.g., statements, methods, classes, Jars, etc. Unlike most prior RTS studies that track changes and dependencies at the same granularity, we use variants of RTS techniques that track changes and dependencies at different granulates. For example, one RTS technique tracks class-level dependencies but computes changes at the Jar level. Tracking changes at the Jar level means that when any class in a Jar changes, the RTS technique select tests as if all classes in the Jar; it trades lower change-computation time for over-selection. The over-selection induced by Jar level change computation is unlikely to make it beneficial for individual projects, but could still be beneficial in large ecosystems because analysis results can be cached. Tracking changes at smaller granularity than the dependency granularity would give the same results as tracking changes at the same granularity as the analysis, because changes need to be projected on the dependencies.

Computing Test Dependencies. Test dependencies can be computed statically or dynamically, at class or project (i.e., Jar) granularity. Class-level dependencies are classes that each test depends on, while Jar-level dependencies computation involves discovering all test-containing Jars that can reach a Jar containing a changed class. Techniques that statically compute test dependencies typically analyze compile-time information to extract a class- or project-level dependency graph. Statically computing dependencies at the Jar level requires knowledge of each project's dependencies. At Facebook, Google and Microsoft, each Jar (also called targets, modules, or nodes) explicitly declares in its build file, all Jars that it depends on. Similarly in Maven Central, each Jar declares its Jar-level dependencies in the `pom.xml` file. By parsing information from these build files, one can statically construct a Jar-level dependency graph. Techniques that dynamically compute class-level test dependencies require to instrument test executions to record all classes that each test invokes as its dependencies. Therefore, in this paper, we compare RTS techniques along three dimensions of test dependency computation: (i) statically at the Jar level, (ii) statically at the class level, and (iii) dynamically at the class level. Note that statically computed project-level dependencies are accurate, as declared in the build files. However, statically computed class-level dependencies can be incomplete as the analysis could miss dependencies that can only be reached via reflection [29].

Computing Affected Tests. RTS involves analyzing dependencies against the changes to compute affected tests. This analysis is always done statically, regardless of the granularity of changes and dependencies, and whether dependencies are computed statically or dynamically. Affected tests refers to the set of tests which have at least one dependency that changed. When dependencies are computed statically from a dependency graph (class or Jar level), reachability analysis on the dependency graph is used to analyze the dependencies against the changes [20], [23], [29]. In our experiments, a test is affected if its node can transitively reach a node for any changed class (or the Jar containing a changed class) in the dependency graph. The intuition for the cost savings obtained from running only affected tests is that code changes typically affect a small portion of the code. Therefore the set of affected tests is typically only a fraction of all tests.

V. Experimental Setup

RTS Techniques Studied. In our RTS experiments we compared project-level and class-level RTS in the Maven Central ecosystem. For brevity, we refer to the RTS techniques that we evaluated as JJ, CC_*st*, CJ_*st*, CC_*dyn* and CJ_*dyn*. The first letter means Class- or Jar-level dependency tracking, and the second letter means Class- or Jar-level granularity for computing changes. *st* means static dependency tracking; *dyn* means dynamic dependency tracking. JJ is similar to the project-level RTS performed at Facebook, Google, and Microsoft. JJ builds Jar-level dependencies from `pom.xml` files and computes changes at the Jar level. CC_*st* and CJ_*st* are two variants of static class-level RTS. They track dependencies at the Class level and compute changes at the Class and Jar level respectively. CC_*dyn* and CJ_*dyn* are two variants of dynamic RTS. They track dependencies at the Class level and compute changes at the Class and Jar level respectively.

Finding Changes. Section IV-A describes how we selected commits for evolving libraries from GitHub. For each Java file changed in a commit, we first had to find which bytecode (`.class`) files in the corresponding Maven Central Jars could have changed. We approximate changed `.class` files as those whose names match the changed Java files, or are inner classes thereof. We may have missed additional classes in the changed Java file which do not share the same name. We projected the class-level changes that we so computed to the Jar level in the following way: if a `.class` file in a Jar changed, we consider all classes in that Jar as changed.

Computing Test Dependencies. We computed test dependencies (i) statically at the Jar level, (ii) statically at the class level, and (iii) dynamically at the class level. Computing test dependencies statically at the Jar level was trivial after constructing the Jar-level dependency graph of Maven Central Jars. Dependencies for a test Jar in the Maven Central graph are simply all GAVs that it can transitively reach. To compute static class-level test dependencies, we used the class firewall algorithm [20] on all Jars for all (i) libraries that we identified in Section IV-A, (ii) clients of these libraries, obtained from our Maven Central graph, and (iii) released test Jars on Maven Central that can transitively reach any of the libraries in our Maven Central graph. For each library, we first find all class-level dependencies for each class and construct a *class-level dependency graph (CLDG)*. Nodes in the CLDG are classes and edges represent uses or inheritance. Next, we combine the library's CLDG with the CLDGs of all its transitive clients plus CLDGs for all test Jars that transitively reach the library's node in the Maven Central graph. Finally, the class-level dependencies of each test class are all classes that the test class can reach in the combined CLDG. Note that, for each library, we only build the CLDG once in memory and then "query" the same graph with various change sets.

We used Ekstazi [28] to collect dynamic class-level test dependencies while running all test classes in all test Jars that reach any of the libraries. For fair comparison with statically computed test dependencies, we only use passing tests. The reason is that a dynamic technique may miss to collect some dependencies for a test execution that fails as those dependencies may not be used before the failure.We ran over 1200000 test classes and over 350000 passed with Ekstazi. The other tests fail for a variety of reasons such as missing platform dependencies. Considering the sheer scale at which we conducted our experiments, it would have been too tedious to set up all the right environments for all tests.

Analyzing Dependencies and Finding Affected Tests. Given a change set, we compute affected tests. For JJ, all tests in a test Jar are affected if the test Jar can reach a Jar containing changed class in the MAVEN Central graph. For statically computed class-level dependencies, we followed the class firewall approach [20], [23], [29]: classes that can reach a changed class form a "firewall" around that changed class, and the test classes in the "firewall" are the affected tests. Therefore, for static class-level dependencies, we compute affected tests as test classes that can reach any changed class in the CLDG. Finally, for dynamic class-level dependencies, a test is affected if the dependency file generated by Ekstazi while running that test contains at least one class that changed. We use Ekstazi in two steps. First, we run Ekstazi on each Jar to collect dynamic dependencies as coverage files (one per test). Then, we intersect changes in each commit with Ekstazi coverage files to obtain the affected tests.

Evaluation Metric. We compute the ratio of tests selected by a class-level RTS technique to the tests selected by JJ:

Definition 1: We define the selection rate as:

SelectionRate-RTS(GAV, δ, Δ) = $\frac{SelectedTests^{RTS}(GAV,\delta)}{SelectedTests^{JJ}(GAV,\Delta)}$

$SelectedTests$ is a function parameterized by a technique $RTS \in \{$JJ, CC_st, CJ_st, CC_dyn, CJ_$dyn\}$ that takes as input a change-set and the GAV that is analyzed for changes. (change granularity is either Class (δ) or Jar(Δ)) and produces a set of tests that are affected by the given change.

VI. SELECTION RESULTS

We discuss the results of our experiments to compare the potential savings of various class-level test selection techniques with a baseline project-level (or, Jar-level) technique (JJ) in the MAVEN Central open-source ecosystem. The JJ technique tracks both dependencies and changes at the level of Jars. We compare JJ with two static RTS techniques (Section VI-A): (i) CC_st computes both dependencies and changes at the class level; (ii) CJ_st computes dependencies at the class level but computes changes at the Jar level. We also compare JJ with two dynamic RTS techniques (Section VI-B): (iii) CC_dyn computes class-level dependencies and class-level changes; (iv) CJ_dyn computes class-level dependencies and Jar-level changes. Finally, we provide some data about the scalability of the evaluated techniques, in terms of the sizes of the graphs that we compute and the time it takes to compute them; these numbers give some sense of how feasible an actual tool that implements these techniques could be (Section VI-C).

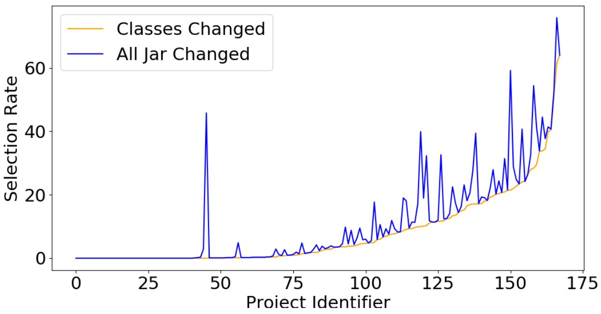

Fig. 5. Selection rate for CC_st and CJ_st

A. Static Class- vs. Jar-Level Dependencies

Figure 5 shows average selection rates for the projects in our experiments, showing the percentage of tests that would be selected by static techniques that track dependencies at the class-level relative to a technique that tracks dependencies at the Jar-level. Specifically, we compare the tests that are selected by CC_st and CJ_st relative to those selected by JJ. The horizontal axis corresponds to project identifiers, and the vertical axis corresponds to selection rates for CC_st (orange line, label: *Classes Changed*) and CJ_st (blue line, label: *All Jar Changed*) relative to JJ for each of the projects in our corpus. The percentages shown are computed over the number of tests that JJ would select. The plot shows some interesting findings. First, it can be observed that a significant number of tests that JJ would select are *not* selected by the class-level techniques. CC_st selects on average across all our projects and changes only 7.8% (SelectionRate-CC_st). CJ_st selects on average across all our projects and changes only 10.5% (SelectionRate-CJ_st). Second, the results also confirm that the percentage of tests selected by CJ_st is always at least as high as the percentage of tests selected by CC_st, for every project. The small difference among all the projects between CC_st and CJ_st demonstrates that the over-approximation of the changes that CJ_st computes is not much larger than the more precise class-level changes computed by CC_st. Therefore, at this scale, it may be profitable to perform selection with changes computed at the Jar level, which also has the added benefit that one may be able to reuse some of reachability computations involved in analyzing the changes.

B. Dynamic Class- vs. Jar-Level Dependencies

Figure 6 shows average selection rates for CC_dyn and CJ_dyn relative to JJ_$|passing$. The horizontal axis corresponds to project identifiers and the vertical axis to selection rates for CC_dyn and CJ_dyn. The labels are the same as in Figure 5. The results show that the CC_dyn selects on average 8.4%, while CJ_dyn selects on average 17.4%. Once

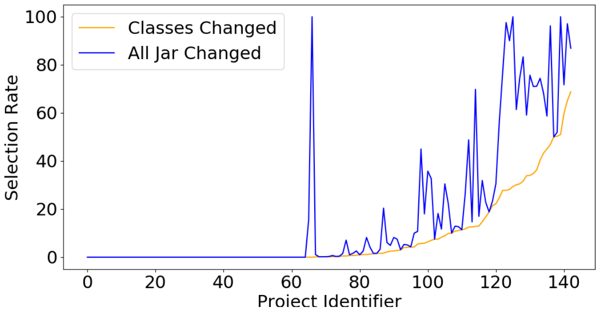

Fig. 6. Selection rate for CC_dyn and CJ_dyn

again, the results suggest that it may be worthwhile to compute changes at the Jar level rather than at the more precise class level, but this time the differences are higher than for the static class-level RTS. These results for the dynamic RTS techniques should not be compared directly with those for the static RTS techniques because the denominator for computing the percentage of tests selected by the dynamic RTS techniques differs from the denominator used to compute the percentages for the static RTS techniques—recall that for dynamic RTS, we only consider tests that passed while collecting coverage with Ekstazi.

C. Size and Timing of Analysis

Because we analyze all clients and track dependencies at the class level, the size of our dependency graphs could become a concern during the analysis. The number of nodes in the dependency graphs varies from 59 to 3142313 (average 279153.7), and the number of edges varies from 103 to 29997498 (average 2321483.6). The size of these graphs affects two computational costs. With our non-optimized implementation, we obtain the following times. First, building the dependency graph can take a considerable amount of time, ranging from 0.2s to 4756.4s (average 167.2s); the time is spent parsing Jar files and class files to extract nodes and edges. Second, computing what tests reach the changed classes ranges from 0.0s to 178.4s (average 3.8s). Each change requires a reachability query of the graph, but for the techniques that track changes at the granularity of Jar, the query can be done only once, and cached for any change to the (library) Jar (as long as the clients do not change); we do not implement this caching and our implementation is quite unoptimized so these times could be much better.

VII. Threats to Validity

Internal. Our scripts that analyze Maven Central, GitHub, perform RTS, and analyze the results produced may have bugs. To mitigate bugs we use mature tools from prior work, e.g., Ekstazi [28], we used assertions in our scripts as sanity checks, e.g., CC_st should select fewer or equal number of tests as CJ_st, we reviewed our scripts, and carefully inspected results.
External. We only analyzed a subset of Maven Central therefore our results may not generalize beyond the subjects and the kinds of changes that we analyzed. To address this, we used over 500K clients for the libraries we analyze; we use 13961 changes across our libraries.
Construct. We analyze tests in Maven Central but many projects do not release test Jars therefore we are underestimating the number of tests that may depend on any given project. However, because our sample of test Jars is rather large, 90703 there is reason to believe that our selection ratios should be similar to those clients and those tests that we did not find in Maven Central as for those clients and tests that we did find in Maven Central.

VIII. Related Work

Studies of Client Breakages Caused by Library Updates. Our formative study is different from prior studies of client breakages caused by library updates in the scale of the ecosystem that we target and the fact that we quantify breakages in *both* compilation and tests. Bavota et al. [37] study the impact of changes to Apache ecosystem libraries on clients in terms of the impacted classes and lines of code. However, they did not quantify compilation or test breakages caused by library changes as we do in our formative study (Section III). de Souza and Redmiles [39] analyze how developers manage dependencies and changes. Their study involves only two teams—i.e., not an ecosystem—and they do not report on breakages that occur when clients upgrade to latest library versions. Bavota et al. [41] consider the impact that unstable APIs have on ratings that Android users assign to apps. Their work is not concerned with test or compilation breakages in ecosystems but shows the impact unstable APIs can have on users. Tufano et al. [46] report on the frequency and reasons behind compilation (not test) failures of historical snapshots in the Apache ecosystem, finding that the most common reason for such compilation failures is missing dependencies. Our formative study reports on both compilation and test failures that occur in releases (i.e., not snapshots) when updating versions of dependencies that are present (i.e., not missing) in the Maven Central ecosystem, with the goal of evaluating regression testing opportunities. McDonnell et al. [40] studied the rates of adoption of API updates by Android ecosystem developers, which showed that developers are slow to adopt new versions of APIs. However, they did not quantify the manner in which apps fail when they adopt new versions of APIs like we do in our formative study.
Previous Studies on RTS. The study presented in this paper is the *largest* study of RTS till date in terms of the number of projects evaluated and also because we consider not just individual projects in isolation but inter-related projects in a very large ecosystem. Many techniques were proposed for RTS [13], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [54]. These techniques compute test dependencies statically or dynamically. RTS techniques that compute dependencies purely statically were proposed in the mid 1990s [20], and we recently showed that, at the class-level, computing dependencies statically can perform comparably with those that compute dependencies dynamically [29]. Vasic

et al. [54] compared class- and module-level RTS *within individual projects*; we compared class- and project-level RTS *across* projects, in a very large ecosystem. We are the first to evaluate statically and dynamically computed dependencies at the class level for RTS at ecosystem scale.

Open-Source Environments and Compatibility. Raemaekers et al. [51] study binary compatibility in MAVEN Central to determine whether binary releases follow semantic versioning, i.e., only major version releases may break compatibility. They found that, in practice, releases do *not* follow semantic versioning. Our work is complementary: we show how library developers could use RTS to cheaper run clients' tests.

Kalra et al. [14] propose POLLUX, a system that advises whether clients can update to a newer library version. POLLUX compares traces and test outcomes from running tests in both the old and new library versions. POLLUX analysis is only from the perspective of the clients; we are concerned with both clients and libraries. Further, our goal is to quantify how much class- vs. project-level RTS can benefit large ecosystems.

Zhou and Walker [15] observed that some very popular libraries sometimes removed some API, then restored it before finally marking it as deprecated. Their hypothesis is that library developers realized after removing the API that some clients were still using the API. It would have been better if these library developers had a way to quicker and cheaper check if removing APIs would break clients.

Mezzetti et al. [55] proposed *type-regression testing* to check that changes to types in library APIs do not break clients in NPM. They found breakages in minor and patch versions of projects that follow semantic versioning. Their work focuses on finding regressions due to changed types in an ecosystem for a dynamically typed language; we study opportunities for RTS to speed up regression testing in an ecosystem for Java.

Studies of Open-Source Environments. Other studies have been based on large repositories containing many project artifacts. Specifically, Vargas-Baldrich et al. [56] proposed `Sally`, a learning-based technique for tagging (assigning keywords to software artifacts) Maven-based projects. They compared the `Sally` results with the set of manually assigned tags for a subset of projects on mavenrepository.com and SourceForge. Also, Mitropoulos et al. [57] have provided a data-set about projects obtained from Maven Central, and show results of statically analyzing these projects with FindBugs. Hilton et al. [58] study the usage of continuous integration (CI) in open-source GitHub projects and find that popular projects use CI, with overall CI usage increasing. Our RTS can enable enhancing CI to run the client tests in addition to the projects' own tests; our results show potential benefits of running class-level vs. project-level RTS. Raemaekers et al. [48] study MAVEN Central and provide many metrics (with regards to size, inter-dependencies, and versions) but do not consider the problem of running tests against clients.

Inter-Project Regression Testing. There has been a lot of work on building systems that perform regression testing across projects in industry. Companies have built practical systems that perform RTS at the equivalent of Jar level, e.g., Facebook's Buck [18], Google's TAP [16], [19], and Microsoft's CloudBuild [17]. CloudBuild addressed several scalability challenges in distributing test runs, caching analyses, and results. Google reported on their pre-submit methodology of testing software to enforce, before merging the changes into the repository, that all tests of all clients pass [32]. Dosinger et al. [59] describe a system to perform regression testing among projects by setting up a network of communicating CI servers that notify dependent projects' CI whenever there is a change and run all the tests in the dependent project. CRAN [4] has a regression testing system which ensures that any new release of a library passes its own tests as well as the tests of all its clients [38]. Similarly, NPM has a tool, called `dont-break` [60], to ensure that new library changes do not break clients [55]. All these systems motivate our research on evaluating RTS opportunities in open-source ecosystems, and especially the opportunities for class-level RTS to improve on the project-level RTS that these systems currently employ.

IX. Conclusion and Future Work

We evaluated RTS opportunities at the scale of MAVEN Central, the largest ecosystem for Java. Our formative study showed that half of the libraries we analyze may benefit from early and frequent running of tests not only in libraries but also in all their clients. We investigated five RTS techniques, spanning several granularities at both the dependency tracking and change computation level, which can reduce the costs to run clients' tests for evolving changes in libraries. The results showed that finer-grained, class-level techniques can select an order of magnitude fewer tests than coarser-grained, project-level techniques currently used in industry: static class-level RTS selects 7.8%–10.5% of tests, and dynamic class-level RTS selects 8.4%–17.4% of tests.

Our empirical study has implications for both practice and research. We show that client-library breakages do occur at the ecosystem level, and we show that finer-grained RTS techniques offer opportunities to make ecosystem-scale RTS more tractable. Our results motivate further research on RTS at scale in very large (open-source) ecosystems, including building practical systems that can work at this scale. Potential future work includes: (1) evaluating the end-to-end time of class-level RTS to analyze changes, select affected tests, and run them rather than just measuring test-selection ratios as done in this paper; (2) investigating techniques for culprit finding—at this scale, it will be critical to efficiently find root cause(s) of test failures, beyond just reporting that tests fail; and (3) investigating opportunities to prioritize or select important clients to test rather than all clients.

X. Acknowledgments

We thank Milos Gligoric for his help with the Ekstazi tool, Stefan Winter for excellent shepherding of this paper, and the anonymous reviewers for feedback on a previous draft. This research was partially supported by the NSF Grant Nos. CCF-1409423, CCF-1421503, and CNS-1646305. We also acknowledge Google and Qualcomm gifts.

References

[1] "Maven Central," https://search.maven.org/.

[2] "npm," https://www.npmjs.com/.

[3] "nuget," https://www.nuget.org/.

[4] "The Comprehensive R Archive Network," https://cran.r-project.org/.

[5] "rubygems," https://rubygems.org/.

[6] "pypi," https://pypi.python.org/.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *STVR*, vol. 22, no. 2, 2012.

[8] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *ICSE*, 2012.

[9] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *ICSE*, 2013.

[10] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades," in *CSMR-WCRE*, 2014.

[11] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *TOSEM*, vol. 24, no. 2, 2014.

[12] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *FSE*, 2014.

[13] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *ESEC/FSE*, 2015.

[14] S. Kalra, A. Goel, D. Khanna, M. Dhawan, S. Sharma, and R. Purandare, "POLLUX: Safely upgrading dependent application libraries," in *FSE*, 2016.

[15] J. Zhou and R. J. Walker, "API deprecation: A retrospective analysis and detection method for code examples on the web," in *FSE*, 2016.

[16] "Testing at the speed and scale of Google," http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[17] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *ICSE SEIP*, 2016.

[18] "Buck," https://buckbuild.com/.

[19] "Tools for continuous integration at Google scale," https://www.youtube.com/watch?v=b52aXZ2yi08.

[20] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *JOOP*, vol. 8, no. 2, 1995.

[21] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *TOSEM*, vol. 6, no. 2, 1997.

[22] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *OOPSLA*, 2001.

[23] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *FSE*, 2004.

[24] G. Xu and A. Rountev, "Regression test selection for AspectJ software," in *ICSE*, 2007.

[25] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: A systematic review," in *ESEM*, 2008.

[26] L. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *IST*, vol. 51, no. 1, 2009.

[27] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *ICSM*, 2011.

[28] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015.

[29] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *FSE*, 2016.

[30] L. Zhang, "Hybrid regression test selection," in *ICSE*, 2018.

[31] "Apache Gump," https://gump.apache.org/.

[32] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *FSE*, 2014.

[33] "Alibaba Druid," https://github.com/alibaba/druid.

[34] "Alibaba Druid Maven Central," http://search.maven.org/\#artifactdetails|com.alibaba|druid|1.0.24|jar.

[35] "Release Java 8 changes to Guava," https://github.com/google/guava/commit/73e382fa877f80994817a136b0adcc4365ccd904.

[36] "Final fields can no longer be set," https://github.com/kohsuke/args4j/commit/6e11f89d40dcc518c0e5eb9eef5d74f05d58e6af.

[37] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: An evolutionary study," *EMSE*, vol. 20, no. 5, 2015.

[38] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: Cost negotiation and community values in three software ecosystems," in *FSE*, 2016.

[39] C. R. B. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *ICSE*, 2008.

[40] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *ICSM*, 2013.

[41] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change- and fault-proneness on the user ratings of Android apps," *TSE*, vol. 41, no. 4, 2015.

[42] C. R. B. de Souza and D. F. Redmiles, "On the roles of APIs in the coordination of collaborative software development," *CSCW*, vol. 18, no. 5, 2009.

[43] R. Holmes and R. J. Walker, "Customized awareness: recommending relevant external change events," in *ICSE*, 2010.

[44] B. E. Cossette and R. J. Walker, "Seeking the ground truth: A retroactive study on the evolution and migration of software libraries," in *FSE*, 2012.

[45] R. Padhye, S. Mani, and V. S. Sinha, "NeedFeed: Taming change notifications by modeling code relevance," in *ASE*, 2014.

[46] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *JSEP*, vol. 29, no. 4, 2017.

[47] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *ICSM*, 2012.

[48] ——, "The Maven repository dataset of metrics, changes, and dependencies," in *MSR*, 2013.

[49] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *MSR*, 2016.

[50] "Semantic Versioning," https://semver.org/.

[51] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," TU Delft, Software Engineering Research Group, Tech. Rep., 2014.

[52] "Learn Git and GitHub without any code!" https://github.com.

[53] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *ICSE*, 2014.

[54] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .NET," in *FSE Industrial Papers*, 2017.

[55] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in Node.js libraries," in *ECOOP*, 2018.

[56] S. Vargas-Baldrich, M. Linares-Vásquez, and D. Poshyvanyk, "Automated tagging of software projects using bytecode and dependencies," in *ASE*, 2015.

[57] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The bug catalog of the Maven ecosystem," in *MSR*, 2014.

[58] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *ASE*, 2016.

[59] S. Dösinger, R. Mordinyi, and S. Biffl, "Communicating continuous integration servers for increasing effectiveness of automated testing," in *ASE*, 2012.

[60] "The dont-break package for npm," https://www.npmjs.com/package/dont-break.