2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Ekstazi: Lightweight Test Selection

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov
University of Illinois at Urbana-Champaign
{gliga,eloussi2,marinov}@illinois.edu

***Abstract*—Regression testing is a crucial, but potentially time-consuming, part of software development. Regression test selection (RTS), which runs only a subset of tests, was proposed over three decades ago as a promising way to speed up regression testing. However, RTS has not been widely adopted in practice. We propose EKSTAZI, a lightweight RTS tool, that can integrate well with testing frameworks and build systems, increasing the chance for adoption. EKSTAZI tracks dynamic dependencies of tests on files and requires no integration with version-control systems. We implemented EKSTAZI for Java+JUnit and Scala+ScalaTest, and evaluated it on 615 revisions of 32 open-source projects (totaling almost 5M LOC). The results show that EKSTAZI reduced the end-to-end testing time by 32% on average compared to executing all tests. EKSTAZI has been adopted for day-to-day use by several Apache developers. The demo video for EKSTAZI can be found at http://www.youtube.com/watch?v=jE8K5_UCP28.**

I. INTRODUCTION

Regression test selection (RTS) is a promising approach to speed up regression testing. Engström et al. [5] and Yoo and Harman [20] present surveys of RTS techniques. The inputs to a traditional RTS technique are two software revisions (new and old), and the dependency information from the test runs on the old revision. The output is a subset of the test suite that can be affected by the changes, and should be re-run in the new revision. RTS is orthogonal to several other techniques used to speed up regression testing (e.g., parallel execution) and can be combined with them for added savings.

While RTS was proposed over three decades ago [5, 20], it has not been widely adopted in practice, except for the Google TAP system [18]. Unfortunately, TAP performs RTS only *across projects* and provides no benefit *within a project*. However, most developers work on one isolated project at a time rather than on a project from a huge codebase as done at Google. Moreover, our recent empirical study [8] shows that developers who work on such isolated projects frequently perform *manual* RTS.

We propose EKSTAZI, a novel RTS tool based on *file dependencies*. EKSTAZI is motivated by recent advances in build systems [1, 7] and prior work on RTS based on class dependencies [4–6, 12, 13, 16, 17] and external resources [10, 11, 15, 19]. Unlike most prior RTS techniques based on finer-grained dependencies (e.g., methods or basic blocks), EKSTAZI does *not* require integration with version-control systems: EKSTAZI does not explicitly compare the old and new revisions. Instead, EKSTAZI computes for each test class what files it depends on; the files can be executable code (e.g., `.class` files in Java) or external resources (e.g., configuration files). A test class need not be run in the new revision if none of its dependent files changed.

We implemented our EKSTAZI tool in Java and integrated it with two popular testing frameworks (JUnit and ScalaTest) and two popular build systems (Maven and Ant). Our tool has many features to support (Java and Scala) projects running on JVMs (e.g., packing of `.class` files in `.jar` archives, instrumentation to collect dependencies using class loaders or Java agents, reflection, etc.).

II. USAGE

Before describing in more detail how EKSTAZI works, we describe in this section how users can integrate EKSTAZI with projects that use Maven or Ant. We also describe programmatic invocation of EKSTAZI that could be used to integrate EKSTAZI with other testing frameworks (e.g., TestNG). EKSTAZI is currently available as a binary [2], and we are preparing sources for release.

A. Integration with Maven

EKSTAZI distribution includes a Maven plugin [2], available from Maven central. Only a single step is required to integrate EKSTAZI with existing build configuration files (i.e., pom.xml); include EKSTAZI in the list of plugins:

```
<plugin>
  <groupId>org.ekstazi</groupId>
  <artifactId>ekstazi-maven-plugin</artifactId>
  <version>${ekstazi.version}</version>
</plugin>
```

where ${ekstazi.version} denotes the version of EKSTAZI.

EKSTAZI offers three optional parameters: (1) `forceall` to force the execution of all tests (even if not affected by changes) and recollect dependencies, (2) `forcefailing` to force the execution of the tests that failed in the previous run (even if not affected by changes), and (3) `skipme` to have all tests run without EKSTAZI.

The same step can be used to integrate EKSTAZI with a Scala project that uses Maven+ScalaTest.

B. Integration with Ant

EKSTAZI distribution also includes an Ant task [2] that can be easily integrated with existing build definitions (i.e., build.xml). The following three steps are required:

1) add namespace definitions to the project element:

```
<project ... xmlns:ekstazi="antlib:org.ekstazi.ant">
```

2) add EKSTAZI task definition:

```
<taskdef uri="antlib:org.ekstazi.ant" resource="org/ekstazi/ant/antlib.xml">
  <classpath path="org.ekstazi.core-${ekstazi.version}.jar"/>
  <classpath path="org.ekstazi.ant-${ekstazi.version}.jar"/>
</taskdef>
```

3) wrap existing JUnit target elements with EKSTAZI select:

```
<ekstazi:select><junit fork="true" ...> ... </junit></ekstazi:select>
```

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.230

C. Programmatic Invocation

Programmatic invocation provides an extension point to integrate EKSTAZI with other testing frameworks (e.g., TestNG). EKSTAZI offers three API calls to check if any dependency is modified, to start collecting dependencies, and to finish collecting dependencies:

```
org.ekstazi.Ekstazi.inst().checkIfAffected("name")
org.ekstazi.Ekstazi.inst().startCollectingDependencies("name")
org.ekstazi.Ekstazi.inst().finishCollectingDependencies("name")
```

where "name" is an id to refer to the collected dependencies for a segment of code (e.g., a fully qualified test class name). These primitives can be invoked from any JVM code. For example, to integrate EKSTAZI with JUnit, we implement a listener that invokes `startCollectingDependencies` before JUnit executes the first test method in a class and invokes `finishCollectingDependencies` after JUnit executes the last test method in a class.

III. TECHNIQUE AND IMPLEMENTATION

A typical RTS technique has three phases: the *analysis (A) phase* selects what tests to run in the current revision, the *execution (E) phase* runs the selected tests, and the *collection (C) phase* collects information for the next revision. EKSTAZI collects dependencies at the level of *files*. We first describe the format in which EKSTAZI stores the dependencies. We next provide more details on each phase and their integration with a testing framework. We finally describe an important optimization to make EKSTAZI practical.

A. Dependency Format

EKSTAZI stores dependencies in a simple format similar to the dependency format of build tools such as Fabricate [7]. For each test class, EKSTAZI stores the names and checksums of the files that the class uses during execution. The checksum hashes the content of the files. These checksums allow EKSTAZI to check changes with no explicit access to the old revision. EKSTAZI stores all the information in a separate *dependency file*[1] for each test class.

B. Analysis (A) Phase

The analysis phase in EKSTAZI is quite simple (and thus fast). For each test class, EKSTAZI checks if the checksums of all dependent files are still the same. If so, the test class is not selected. This check requires no sophisticated comparisons of the old and new revisions (which prior RTS research techniques usually perform on the source), and in fact it does not even need to analyze the old revision (much like a build system can incrementally compile code just by knowing which source files changed).

EKSTAZI naturally handles newly added test classes: if there is no dependency information for some class, it is selected. Initially, on the very first run of EKSTAZI, there is no dependency information for any class, so they are all selected.

[1]Note that a "dependency file", which stores dependencies, should not be confused with "dependent files", which are the dependencies themselves.

C. Execution (E) Phase

We integrated EKSTAZI with JUnit because it is a widely used framework for executing unit tests in Java. Although one can initiate test execution directly from JUnit, large projects typically initiate test execution from a build system (e.g., Maven or Ant). We describe integration of Ekstazi in this typical scenario.

EKSTAZI first determines what test classes *not* to run. This avoids the unnecessary overhead (e.g., loading classes or spawning a new JVM) of preparing to run a class and finding it should not run. The A phase makes an `excludes` list of test *classes* that should not run, and the build system ignores them before executing the tests.

There are two possible approaches to integrate the E and C phases. The first and simplest way is to do it in *one pass*. The dependencies for the test classes that were not selected cannot change. However, the test classes that were selected need to be run to determine if they still pass or fail, and thus to inform the user who initiated the test session. Because the dependencies for these classes changed, the simplest way to update their dependency files is with one pass that both determines the test outcome and updates the dependency files. However, collecting dependencies has an overhead. Therefore, some settings may prefer to use *two passes*: one without collecting dependencies, just to determine the test outcome and inform the user as fast as possible, and another to also collect the dependencies. The second pass can be started in parallel with the first or can be performed sequentially later.

D. Collection (C) Phase

The collection phase creates the dependency files for the executed test classes. EKSTAZI monitors the execution of the tests and the code under test to collect the set of files accessed during execution of each class, computes the checksum for these files, and stores them in the corresponding dependency file. EKSTAZI currently collects *all* files that are either read or written, but it could be even more precise to distinguish writes that do not create a dependency [9]. Moreover, EKSTAZI tracks even files that were attempted to be accessed but did not exist; if those files are added later, the behavior can change.

In principle, we could collect file dependencies by adapting a tool such as Fabricate [7] or Memoize [14]: these tools can monitor any OS process to collect its file dependencies, and thus could be used to monitor a JVM that runs tests. However, these tools would be *imprecise* for two reasons. First, they would not collect dependencies per test class when multiple test classes run in one JVM. Second, they would not collect dependencies at the level of `.class` files archived in `.jar` files. Moreover, these tools are not portable from one OS to another, and cannot be easily integrated with testing frameworks (e.g., JUnit) and build systems (e.g., Maven).

To precisely collect accessed files, EKSTAZI dynamically instruments the bytecode and monitors the execution to collect both explicitly accessed files (through the `java.io` package) and implicitly accessed files (i.e., the `.class` files that contain the executed bytecode). EKSTAZI collects explicitly used files

by monitoring all standard Java library methods that may open a file (e.g., `FileInputStream`). Files that contain bytecode for Java classes are not explicitly accessed during execution; instead, a class loader accesses a classfile when a class is used for the first time. Our instrumentation collects a set of objects of the type `java.lang.Class` that a test covers during execution; EKSTAZI then finds for each class where it was loaded from. If a class is not loaded from disk but dynamically created during execution, it need not be tracked as a dependency.

More precisely, EKSTAZI instruments the following code points: (1) start of a constructor, (2) start of a static initializer, (3) start of a static method, (4) access to a static field, (5) use of a class literal, (6) reflection invocations, and (7) invocation through `invokeinterface` (bytecode instruction). EKSTAZI needs no special instrumentation for test classes: they get captured as dependencies when their constructor is invoked by JUnit. EKSTAZI also does not instrument the start of instance methods: if a method of class `C` is invoked, then an object of class `C` is already constructed, which captured the dependencies on `C`.

E. Non-debug Checksums

EKSTAZI's use of file checksums offers several advantages, most notably (1) the old revision need not be available for the A phase, and (2) hashing to compute checksums is fast. On top of collecting the executable files (`.class`) from the archives (`.jar`), EKSTAZI can compute the *non-debug checksum* for the `.class` files. Computing the checksum from bytecodes already ignores some changes in the source code (e.g., `i++` and `i+=1` could be compiled the same way). The base approach computes the checksum from the entire file content, including all the bytecodes. However, two somewhat different executable files may still have the same semantics in most contexts. For example, adding an empty line in a `.java` file would change the debug info in the corresponding `.class` file, but almost all test executions would still be the same (unless they explicitly observe the debug info). EKSTAZI can ignore certain file parts, such as compile-time annotations and other debug info, when computing the checksum. The trade-off is that the non-debug checksum makes the A and C phases slower (rather than quickly applying a hashing function on the entire file, EKSTAZI needs to parse the file and run the hashing function on parts of it), but it makes the E phase faster (as EKSTAZI selects fewer tests).

IV. EVALUATION

This section (1) describes the projects used in our experimental evaluation of EKSTAZI, (2) describes the experimental setup, and (3) reports the RTS results in terms of the number of selected test classes and the end-to-end time.

A. Projects

We used 32 open-source projects (totaling 4,937,189 LOC) to evaluate EKSTAZI. The set of projects was chosen by three undergraduate students who were not familiar with our study. We suggested starting places with larger open-source projects: Apache Projects, GitHub, and GoogleCode. We also asked that each project satisfies several requirements: (1) has the latest available revision build without errors, (2) has at least 100 JUnit tests, (3) uses Maven or Ant, and (4) uses SVN or Git. The first two requirements were necessary to consider compilable, non-trivial projects, but the last two requirements were set to simplify automation of the experiments.

From about 100 projects initially considered, two-thirds were excluded because they did not build (e.g., due to syntax errors or missing dependencies), used a different build system (e.g., Gradle), or had too few tests. The students confirmed that they were able to execute JUnit tests in all selected projects.

We performed our evaluation on 615 revisions of 32 projects. To the best of our knowledge, this is the largest dataset used in any RTS evaluation outside of Google [3].

B. Experimental Setup

The goal is to evaluate how EKSTAZI performs if RTS is run for each committed project revision. (In general, developers may run RTS even between commits [8], but there is no dataset that would allow *executing* tests the same way that developers executed them in between commits.) For each project, our script checks out the revision that is 20 revisions *before* the latest revision available at the time of the first download. If any revision cannot build, it is ignored. If it can build, the script executes the tests in three scenarios: (1) RetestAll executes all tests in JUnit (without EKSTAZI), (2) executes the tests with EKSTAZI while collecting dependencies in all AEC phases (the way that a developer would use the tool), and (3) executes the tests with EKSTAZI but without collecting dependencies, only the AE phases (for the sake of experiments). The script then repeats these steps for all revisions until reaching the latest available revision.

In each step, the script measures the number of executed tests—(1) all tests for JUnit or (2&3) selected tests for EKSTAZI—and the testing time—(1) the execution time of all tests for JUnit, (2) the end-to-end time for all AEC phases of EKSTAZI, or (3) just the time for the AE phases of EKSTAZI. The script measures the times to *execute the build command* that the developers use to execute the tests (e.g., `mvn test` or `ant junit-tests`). Finding the appropriate command took a bit of effort because different projects use different build target names, or the entire test suites for the largest projects run too long to perform our experiments on multiple revisions in reasonable time. We sometimes limited the tests to a part of the entire project (e.g., the `core` tests for Hadoop in RetestAll take almost 8 hours across 20 revisions, and the full test suite takes over 17 hours for just one revision). By measuring the time for the build command, we evaluate the speedup that the developers would have observed had they used EKSTAZI. Note that the speedup that EKSTAZI provides over RetestAll is even bigger for the testing itself than for the build command, because the build command has fixed overhead before initiating the testing.

Project	Link at https://github.com/
Apache Camel	apache/camel
Apache Commons Math	apache/commons-math
Apache CXF	apache/cxf
Camel File Loadbalancer	garethahealy/camel-file-loadbalancer
JBoss Fuse Examples	garethahealy/jboss-fuse-examples
Jon Plugins	garethahealy/jon-plugins
Zed	hekonsek/zed

Fig. 1: Current EKSTAZI users

C. Summary of Results

We ran all experiments on a 4-core 1.6 GHz Intel i7 CPU with 4GB of RAM, running Ubuntu Linux 12.04 LTS. We used three versions of Oracle Java 64-Bit Server: 1.6.0_45, 1.7.0_45, and 1.8.0_05 because different versions were necessary as several projects require specific older or newer Java version. The testing time is the key metric to compare RetestAll, EKSTAZI AEC, and EKSTAZI AE runs; as an additional metric, we use the number of executed tests.

Overall, the selection ratio of test classes varies between 5% and 38% of RetestAll, the time for AEC varies between 9% and 138% (where over 100% is slowdown), and the time for AE varies between 7% and 99%. On average, across all the projects, the AEC time is 68%, and the AE time is 53%. More importantly, all slowdowns are for projects with short-running test suites (i.e., less than one minute). Considering only projects with long-running test suites, the AEC time is 46%, and the AE time is 34%.

D. Case Study: Apache CXF

EKSTAZI has already been adopted by several open-source projects listed in figure 1. We evaluated how EKSTAZI performed on one of these projects (Apache CXF) over 80 selected recent revisions, after EKSTAZI was included in the project. Figure 2 shows how EKSTAZI compares with RetestAll in terms of end-to-end time. The plot shows that EKSTAZI brought substantial savings to Apache CXF.

V. CONCLUSIONS

We described EKSTAZI, a tool for regression test selection. EKSTAZI collects file dependencies for each test class, and detects affected tests by checking if their dependent files changed. EKSTAZI is easy to use and integrates with popular build systems. Moreover, our evaluation shows promising results that can increase the chance of RTS adoption in practice. Several open-source projects have already adopted EKSTAZI.

ACKNOWLEDGMENTS

We thank Alex Gyori, Farah Hariri, Owolabi Legunsen, Yu Lin, Qingzhou Luo, Aleksandar Milicevic, and August Shi for feedback on this work, and Dan Schweikert, Rohan Sehgal, Nikhil Unni, and Andrey Zaytsev for helping with the experiments in the evaluation. This material is based upon work partially supported by the NSF Grant Nos. CNS-0958199, CCF-1012759, CCF-1421503, and CCF-1439957.

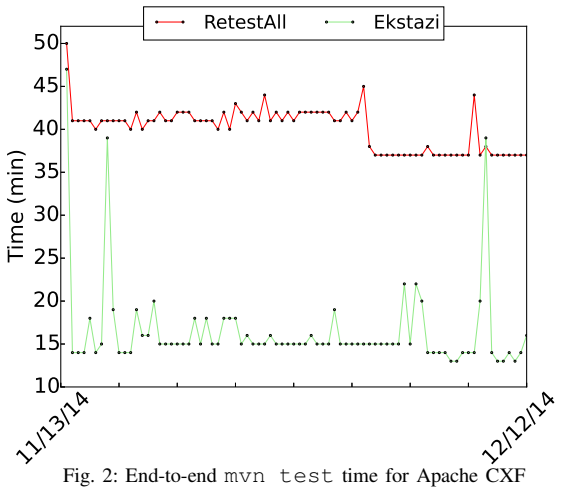

Fig. 2: End-to-end `mvn test` time for Apache CXF

REFERENCES

[1] Build in the cloud. http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.

[2] Ekstazi. http://ekstazi.org/.

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

[4] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *PROFES*. 2010.

[5] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *I&ST-J*, 52(1), 2010.

[6] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM*, 2008.

[7] Fabricate. https://code.google.com/p/fabricate/.

[8] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ASE*, 2014.

[9] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *ISSTA*, 2011.

[10] R. A. Haraty, N. Mansour, and B. Daou. Regression testing of database applications. In *SAM*, 2001.

[11] R. A. Haraty, N. Mansour, and B. Daou. Regression test selection for database applications. *ATDR*, 3, 2004.

[12] P. Hsia, X. Li, D. Chenho Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *JSM*, 9(4), 1997.

[13] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2), 1995.

[14] Memoize. https://github.com/kgaughan/memoize.py.

[15] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *ICST*, 2011.

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

[17] M. Skoglund and P. Runeson. Improving class firewall regression test selection by removing the class firewall. *JOOP*, 17(3), 2007.

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

[19] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *ICSM*, 2005.

[20] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2), 2012.