STARTS: *STA*tic *R*egression *T*est *S*election

Owolabi Legunsen, August Shi, Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
Email: {legunse2,awshi2,marinov}@illinois.edu

***Abstract*—Regression testing is an important part of software development, but it can be very time consuming. Regression test selection (RTS) aims to speed up regression testing by running only impacted tests—the subset of tests that can change behavior due to code changes. We present STARTS, a tool for *STA*tic *R*egression *T*est *S*election. Unlike dynamic RTS, STARTS requires no code instrumentation or runtime information to find impacted tests; instead, STARTS uses only compile-time information. Specifically, STARTS builds a dependency graph of program types and finds, as impacted, tests that can reach some changed type in the transitive closure of the dependency graph. STARTS is a Maven plugin that can be easily integrated into any Maven-based Java project. We find that STARTS selects on average 35.2% of tests, leading to an end-to-end runtime that is on average 81.0% of running all the tests. A video demo of STARTS can be found at https://youtu.be/PCNtk8jphrM.**

I. Introduction

Regression testing [1] is an important part of software development. After every code change, a developer runs the tests in the regression test suite to ensure that the changes do not break any existing functionality. However, when a regression test suite contains many tests, running all tests after every change, often called *RetestAll*, is very time consuming and slows down the software development process. Companies such as Google and Microsoft have reported how expensive it is for them to perform regression testing [2]–[7]. Regression test selection (RTS) is a way to reduce the cost of regression testing by selecting to run only the tests *impacted* by the changes [1]. An RTS technique works by finding the dependencies of each test and selecting the tests that depend on the changes. Running fewer, but necessary, tests speeds up regression testing, while aiming not to miss any test failures.

In our prior work [8], we used a prototype static RTS tool to compare with dynamic RTS, which computes test dependencies dynamically. The results showed that static RTS with dependencies computed at the class-level is comparable with the state-of-the-art dynamic RTS tool Ekstazi [9], [10]. The results are encouraging, showing that static RTS is feasible, and worthy of further research. Performing RTS statically could be particularly useful in contexts where dynamic RTS is not feasible, such as when dynamic instrumentation to collect test dependencies breaks time-sensitive tests or when non-determinism causes dynamic RTS to collect wrong or incomplete test dependencies.

We present STARTS (*STA*tic *R*egression *T*est *S*election), a robust tool for performing static RTS. STARTS constructs a dependency graph relating all types (including classes, interfaces, and enums) in an application and computes a transitive closure for each test to find its dependencies. STARTS determines the types that changed by computing the checksum of each type's corresponding compiled classfile and comparing the computed checksum with the one computed in the prior run. STARTS selects to run impacted tests, which are tests whose transitive dependencies include some changed type. We made several changes to our initial research prototype [8] to make STARTS robust and usable on real, large software projects: we added support for multi-module Maven projects and improved the speed, including parsing constant pools instead of entire classfiles, saving dependencies as *type-to-tests* instead of *test-to-types*, using a faster graph library (yasgl [11], instead of JGraphT [12]), and adding incremental caching of dependencies.

We evaluated STARTS on 32 Maven-based projects from GitHub. We find that STARTS selects on average 35.2% of all tests, leading to an end-to-end runtime (consisting of the time to select what tests to run plus time to run those tests) that is 81.0% of the RetestAll time to run all tests. STARTS scales well, and for 11 projects with longer-running tests that take over one minute to run, STARTS selects on average 40.5% of all tests, leading to an end-to-end runtime that is only 68.2% of the RetestAll time. STARTS source code is publicly available on GitHub at https://github.com/TestingResearchIllinois/starts, and binary code is released on Maven Central.

II. Usage

STARTS is a Maven plugin [13] and can be easily integrated with any Maven-based Java project.

Integrating STARTS: The easiest way to integrate STARTS with a project is to add the latest version of the STARTS plugin from Maven Central to the project's `pom.xml` file:

```
<plugin>
  <groupId>edu.illinois</groupId>
  <artifactId>starts-maven-plugin</artifactId>
  <version>${latest_STARTS_version}</version>
</plugin>
```

Using STARTS: Developers can use STARTS to perform several RTS-related tasks: (i) finding the types that changed, (ii) finding the types (not just tests) that are impacted by the changes (i.e., change-impact analysis), (iii) finding the tests that are impacted by the changes without running those tests, and (iv) finding and running the tests that are impacted by the changes. STARTS provides several goals:

```
$ mvn starts:help # list all goals
$ mvn starts:diff # find types that changed semantically
$ mvn starts:impacted # find types impacted by changes
$ mvn starts:select # find (but not run) impacted tests
$ mvn starts:starts # find and run impacted tests
$ mvn starts:clean # delete STARTS artifacts
```

The first goal, `starts:help`, lists all the goals in STARTS and what they can be used for. The other five goals are related to RTS. `starts:diff` displays all the `Java` types (including classes, interfaces, and enums) that changed STARTS was run. `starts:impacted` displays all types (not just *test* classes) that are impacted by the changes, thereby providing a means for change-impact analysis. `starts:select` displays, but does not run, the test classes that are impacted by the changes since the last time STARTS was run—allowing developers more flexibility to first select impacted tests and then run those tests later. `starts:starts` runs the impacted tests; it performs the functions of the previous RTS-related goals, plus executing the impacted tests. Finally, `starts:clean` removes all artifacts that STARTS stored from a previous run (in a `.starts` directory), resetting STARTS so that in the next run, all types are considered changed (and all tests are selected to be run, if using `starts:starts`).

STARTS provides several options that give some flexibility to the user. The most important option is whether or not to update the STARTS artifacts after invoking a goal. As described in Section III-B, STARTS keeps track of the checksums of all types from the previous run, storing them to disk and using them in the new run to find impacted tests. All RTS-related goals in STARTS provide an `update*Checksums` option, which, when `true`, updates the stored checksums after a run. This option is set to `true` by default for the goal `starts:starts`, but is by default `false` for all other goals.

III. Technique and Implementation

We describe the technique implemented in STARTS and the STARTS Maven plugin.

A. Technique

STARTS performs static RTS (SRTS) at the class level—tests are selected at the test-class (not test-method) level and the dependencies of these tests are also computed at the class/-type level. Our recent work [8], showed that such class-level SRTS outperformed method-level SRTS and was comparable with the state-of-the-art class-level dynamic RTS technique Ekstazi [9]. Thus, we implemented STARTS to perform class-level SRTS based on the idea of a *class firewall* [14]–[16], which encloses the types that need to be retested because they may be impacted by a code change. The class firewall is computed on a *type-dependency graph (TDG)* where each node is a type in the application and there is a directed edge from one type τ to another type τ' if τ has a direct use or inheritance dependency on τ'. Test class nodes are also included in the TDG. If τ_c is a type that changed, then τ is impacted by the change to τ_c iff $\langle \tau, \tau_c \rangle \in E^*$, where E is the set of all edges in the TDG, and * denotes the reflexive and transitive closure. The class firewall is the set of all types that can transitively reach any of the types that changed in the TDG, and can therefore be defined as $firewall(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^*$, where $\mathcal{T}_c$ is the set of all types that changed, $^{-1}$ denotes the inverse relation, and $\circ$ denotes relation composition. Given (1) classfiles for all types (obtained from compiling a new revision) of an application and (2) checksum of classfiles from a prior revision, STARTS can output the set of changed types ($\mathcal{T}_c$), the class firewall ($firewall(\mathcal{T}_c)$), and T_i, the set of impacted tests—computed as the set difference between the set of all tests in the new revision and the set of tests that are not in $firewall(\mathcal{T}_c)$ (which is computed from the old revision). We compute T_i this way so that it includes any newly-added tests while still using the TDG computed on the old revision.

B. Implementation

Figure 1 shows the STARTS architecture, which comprises components to: (i) find dependencies among types in the application, (ii) construct the TDG, (iii) find the changed types between two revisions of the application, (iv) store checksums of all types from the current revision, (v) select the tests impacted by the changed types, and (vi) run the impacted tests.

Finding Dependencies Among Types: STARTS needs to compute the dependencies among all types in the application. The prototype in our prior work [8] used ASM to parse *all* the bytecode in the compiled classfile of a given type in order to compute its dependencies. However, parsing entire classfiles just to find dependencies is rather slow because it requires to recursively visit each type's fields, methods, signatures, and annotations to collect all the types that are referenced. STARTS improves on computing dependencies among types by only reading the constant pool in each classfile to determine all types that the type in the classfile may depend on. We use the recent Oracle *jdeps* tool [17], now part of the standard Java library, to read the constant pools. After the new revision of an application has been compiled to produce classfiles, STARTS makes a single jdeps invocation (via the jdeps API) to parse all classfiles in the application at once, and then processes the jdeps output in memory to find the dependencies for each type.

Constructing the Dependency Graph: The TDG contains an edge from one type to each of its dependencies. We use a custom graph library called yasgl [11] to construct graphs and to find tests that can transitively reach some changed type. We add each type as a node in a yasgl graph and add dependencies computed by jdeps as edges between nodes in the graph. With a yasgl graph, STARTS computes the transitive closure of each test class to find all types that each test depends on. Our initial prototype [8] used JGraphT [12], but yasgl is faster for computing the transitive closure. For example, yasgl takes 1.4 sec to compute the transitive closure for a graph with 41,960 nodes and 509,946 edges (coming from a single module of a project with 110 test classes). JGraphT takes 2.7 sec to compute the same transitive closure, a difference that accumulates when considering all the modules in the project. Note that the yasgl TDG that STARTS uses does

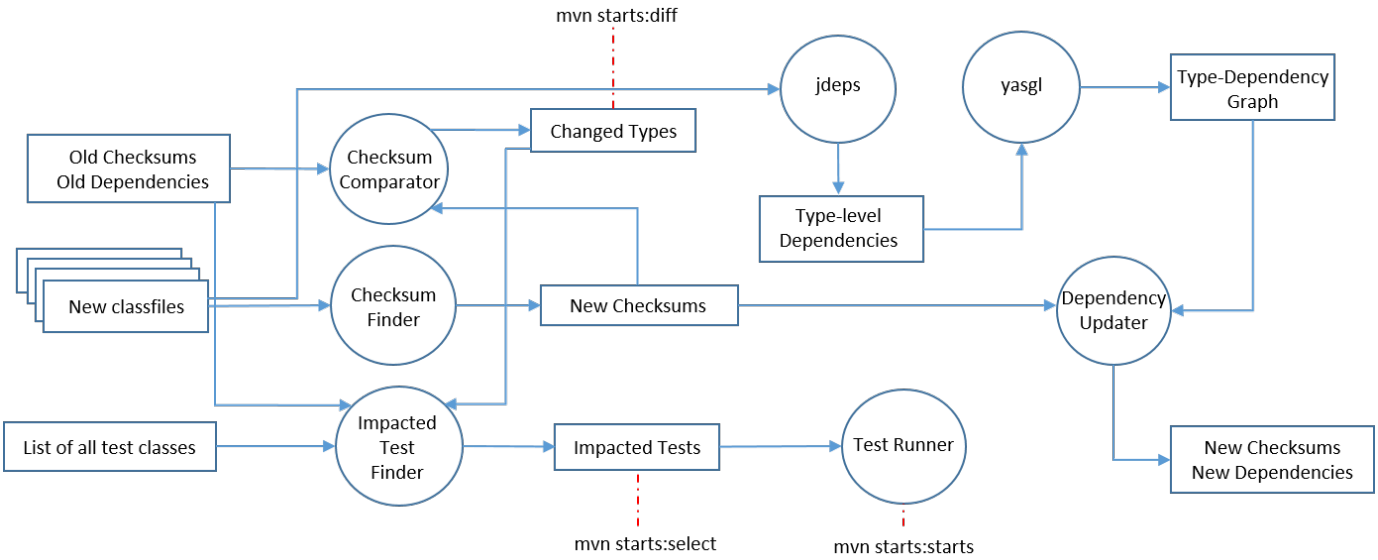

Fig. 1: STARTS Architecture

not distinguish between use edges and inheritance edges, as done in the *Intertype Relation Graph (IRG)* used in our initial prototype and in prior work [16].

Finding Changed Types: STARTS finds the types that changed since the last time it was run. STARTS uses the same checksum function from Ekstazi [9], [10] to compute a checksum that ignores debug-related information for each classfile and stores that checksum to a file. STARTS tracks changes in classfiles because the corresponding source file can be different yet result in the same classfile that is actually executed, so tracking classfiles is more precise. Also, STARTS uses checksums for checking whether a classfile is modified instead of seemingly faster methods like timestamps, which can be unreliable (e.g., Maven's incremental build system is broken [18] and often recompiles every type on each run, so one cannot rely on the timestamps of the classfiles). Once compilation is complete in the new revision, STARTS computes the checksums of all compiled classfiles and compares against the stored checksums computed from the previous revision for each file. If the old and new checksums differ, STARTS considers that type to have changed. If the type had no previously computed checksum (i.e., a new type was added), its checksum is stored for future runs. Finally, if a type for which STARTS previously computed a checksum cannot be found in the new revision (i.e., an old type was deleted), then that type is no longer stored in the checksum file for future runs. If there is no checksum file on disk (e.g., on the very first run, or after running `mvn starts:clean`), STARTS considers all types as changed.

Computing and Storing Checksums: In our initial prototype [8], as well as in Ekstazi, the transitive closure of each test class in the graph was stored as a mapping from each test class to its dependencies, i.e., a *test-to-types* mapping. Further, there was one dependency file per test. Once a tool computed the set of types that changed, it then checked the dependency file of each test to see if the test depends on any of the changed types. However, we observed that STARTS discovers many more test dependencies than Ekstazi, due to inherent imprecision of static analysis, and that many tests shared a lot of these dependencies. As a result, we reversed the dependency storage format in STARTS to reduce the amount of repetitive checking of test dependencies by storing a *type-to-tests* mapping. STARTS stores in a single file a mapping from each type in the application to the set of tests that depend on that type. This file is stored in a directory called `.starts` under the root directory of the application. More precisely, if the application is a multi-module Maven-based project, STARTS creates multiple `.starts` directories, each with its own type-to-tests-mapping file, under each module, and the types may span across modules if that is where the dependencies lead. Updating the checksums that are stored on disk after invoking a STARTS goal on a new revision can be turned on or off, as described in Section II.

The type-to-tests storage format that STARTS uses, together with processing only one file on disk, greatly improves the performance of selecting impacted tests. For example, in one project, STARTS takes 22.9 sec to check if any of the dependencies changed when using the type-to-tests, single-file format, but the same check takes 79.8 sec with Ekstazi's test-to-types, multiple-files format. One possible modification of the type-to-tests format could be to first read all the files and then reverse the mapping (in memory) to be from type to tests before comparing checksums. However, this modification would still incur the cost of reading potentially many files from disk, and it would put the mapping-reversal process on the critical path from when testing is initiated until developers obtain test results—mapping reversal in STARTS can happen in a separate offline phase that is not on the critical path.

Selecting Impacted Tests: STARTS uses the type-to-tests dependency mapping from the previous revision and the set of all changed types to find the tests that are **not** impacted by changes. STARTS then computes the impacted tests as the difference between the set of all tests in the current revision and the set of non-impacted tests. Thus, newly added tests are always in the set of impacted tests. Dependency graph construction on the new revision is *not* required to find impacted tests (allowing quicker computation of impacted tests). Rather, STARTS reads the type-to-tests dependency file which was computed based on the dependency graph constructed in the previous revision. The fact that STARTS requires only compile-time information to find impacted tests can allow a clean separation of phases: an *analysis phase (*a*)* finds changes and impacted tests, an *execution phase (*e*)* runs the impacted tests, and a *graph computation phase (*g*)* builds the dependency graph and uses it to create a type-to-tests mapping for the next revision[1]. This separation can enable the choice to run STARTS in an "online mode" (the *a*, *e*, and *g* phases are run together) or an "offline mode" (the *a* and *e* phases can run separately from or in parallel with the *g* phase). We did not yet implement goals to toggle the online/offline modes, but report times for offline mode as the time for online mode minus the time for the *g* phase. `starts:select` displays the impacted tests but does not run them.

Running Impacted Tests: STARTS computes the set of selected tests to run as previously described: it excludes non-impacted tests from the set of all tests in the application. Specifically, STARTS dynamically adds the non-impacted tests to the set of tests that Surefire plugin is already configured to not run. As a result, when STARTS invokes the Maven Surefire plugin to run the tests, Surefire will run only the tests that are impacted by the changes. The goal `starts:starts` will perform all the previous steps to find changed types, select impacted tests and run those selected tests.

C. Important STARTS Options

STARTS provides a number of other options, in addition to turning on/off the checksum file updates (Section II).

Caching jdeps output: One consideration in the design of STARTS is how to handle the output of running jdeps on third-party libraries (JARs). Many projects do not frequently change their library versions, and using jdeps to parse the library code on each revision would needlessly repeat work. STARTS therefore provides options to (i) use a preprocessed cache, (ii) incrementally build the cache on each revision, and (iii) parse the third-party libraries on each revision. The default is to incrementally build the cache on each revision. When STARTS encounters a JAR in the application's classpath, it first checks whether a corresponding jdeps output file exists in the `jdeps-cache` directory, which is found in each module of the application. If there is one, STARTS reads it; otherwise, STARTS runs jdeps on the JAR, uses the jdeps output for its current processing, and stores the jdeps output for that JAR in a file in the `jdeps-cache` directory. The names of the files in the `jdeps-cache` directory match the Group/ArtifactId/Revision convention of naming Maven-based projects [19], and have a `.graph` extension.

If there is a cache (possibly computed elsewhere or even from different applications), STARTS can be configured to specify the location of this cache. The following command shows using an RTS-related goal with a preprocessed cache, where `${GRAPH_CACHE}` is the directory containing the preprocessed jdeps output for each third-party library and the jdeps output files are organized as described for the default option:

```
$ mvn starts:starts -DgCache=${GRAPH_CACHE}
```

If no cache is input or the cache is empty, STARTS runs jdeps on all libraries on each revision.

File Formats for Checksums and Dependencies: STARTS supports two formats for storing the checksums of all types in the application and the tests that transitively depend on them: the new type-to-tests (`ZLC`) format and the old test-to-types (`CLZ`) format (proposed in Ekstazi). Section III-B describes these formats and their tradeoffs. `ZLC` is the default file format that STARTS uses. To run STARTS using the `CLZ` file format:

```
$ mvn starts:starts -DdepFormat=CLZ
```

Controlling STARTS Artifact Storage: Configuring different logging levels can control the amount of information that STARTS stores between runs, where the logging levels are the same as in the `java.util.logging` API. When running at the default logging `Level.INFO`, STARTS stores only the checksum and dependency file, `.starts/deps.zlc`, between runs. At `Level.FINEST`, STARTS will store all its files: the lists of all/impacted/non-impacted tests, the dependencies that jdeps computed, the classpath that STARTS used, the yasgl graph that STARTS constructed internally, and the set of changed types. Running at logging level `Level.FINER` will store only `.starts/deps.zlc`, the set of impacted tests, and the set of all tests. To run STARTS while storing all its files:

```
$ mvn starts:starts -DstartsLogging=FINEST
```

IV. Evaluation

We ran all experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 16.04.3 LTS and Oracle Java 64-Bit Server version 1.8.0_144. We evaluated STARTS on 32 Maven projects. These projects include 21 single-module Maven projects we used in our previous study [8] and 11 multi-module Maven projects that we did not evaluate before, showing that STARTS can be integrated into larger Maven projects. We ran STARTS on each project over a number of revisions and measured the number of impacted tests that STARTS selected to run, relative to the number of all tests. We also measured the percentage of end-to-end time taken by STARTS relative to the end-to-end time for running all tests, i.e., RetestAll. The STARTS end-to-end time includes the time to compile, perform selection,

[1]The *g* phase in STARTS is analogous to the coverage-collection (*c*) phase in Ekstazi and other dynamic RTS techniques where separation of *c* and *e* phases is harder to achieve in practice.

TABLE I: Statistics about selected tests and end-to-end time of STARTS compared to RetestAll

Project	SHAs [#]	ALL	Selected [#]	Selected [%]	RTA[s]	Offline Time [%]	Online Time [%]	Beakdown			
								a	*e*	*g*	Comp.
headius/invokebinder	68	2.1	1.6	76.0	3.3	110.7	134.8	0.0	20.0	20.0	60.0
google/compile-testing	32	7.3	3.1	44.1	4.4	118.3	137.6	0.0	18.3	13.3	68.3
apache/commons-cli	52	23.0	10.2	44.1	4.8	109.4	126.1	0.0	10.3	10.3	79.3
logstash/logstash-logback-encoder	45	18.2	3.7	23.5	5.6	112.8	129.8	0.0	13.7	12.3	74.0
apache/commons-dbutils	15	24.6	8.2	33.0	5.6	109.1	121.6	0.0	13.0	13.0	73.9
apache/commons-validator	22	61.0	13.8	22.6	6.6	93.5	107.2	0.0	17.1	11.4	71.4
apache/commons-fileupload	8	12.0	3.8	31.2	6.8	98.2	102.1	0.0	12.9	5.7	81.4
apache/commons-codec	65	47.4	2.1	4.5	9.3	69.5	73.9	0.0	10.1	7.2	82.6
srt/asterisk-java	47	38.1	2.3	6.0	9.6	70.2	79.4	0.0	18.4	10.5	71.1
apache/commons-functor	20	164.0	23.2	14.1	10.7	91.7	96.3	0.0	8.7	5.8	85.6
apache/commons-compress	12	89.4	23.8	26.6	13.3	79.8	83.5	0.0	25.9	4.5	69.6
apache/commons-email	10	18.0	5.4	30.0	16.2	68.0	71.9	0.0	39.7	6.0	54.3
square/retrofit	13	32.2	10.3	32.2	21.1	79.7	86.8	5.9	43.0	9.7	41.4
apache/commons-lang	63	133.7	42.8	32.0	24.8	73.3	76.8	0.0	35.3	4.7	60.0
apache/commons-collections	12	164.0	7.2	4.4	25.3	58.3	58.9	0.0	10.0	1.3	88.7
AdoptOpenJDK/jitwatch	23	26.0	10.6	40.6	26.4	58.4	60.9	0.0	62.5	4.4	33.1
graphhopper/graphhopper	8	106.8	70.1	65.7	29.8	90.8	97.3	0.0	45.2	6.9	47.9
apache/commons-imaging	89	58.8	21.5	37.9	29.5	65.2	67.5	0.0	51.5	3.5	45.0
cloudera/oryx	17	58.0	17.3	29.8	37.6	85.0	91.3	6.0	40.1	6.6	47.3
robovm/robovm	11	32.0	9.1	28.4	39.5	107.5	111.6	1.4	5.7	3.6	89.3
ninjaframework/ninja	6	102.0	55.0	53.9	40.5	93.6	120.3	7.2	42.0	22.5	28.3
Average(SHORT)	30.4	58.0	**16.4**	**32.4**	17.6	**87.8**	**96.9**	1.0	25.9	8.7	64.4
apache/commons-math	63	449.9	42.4	9.4	98.3	28.9	30.3	0.3	36.8	4.7	58.2
addthis/stream-lib	7	24.0	5.4	22.6	106.4	47.5	48.4	0.0	88.8	2.1	9.1
apache/commons-io	13	99.2	23.4	23.5	132.0	43.4	43.9	0.0	85.0	1.2	13.8
brettwooldridge/HikariCP	18	26.4	22.4	84.7	132.9	95.2	96.6	0.8	92.9	1.5	4.8
opentripplanner/OpenTripPlanner	9	136.0	76.4	56.2	179.3	82.3	85.4	1.8	83.9	3.7	10.5
undertow-io/undertow	28	220.1	151.5	68.8	181.0	80.5	82.9	1.1	82.4	2.9	13.6
Graylog2/graylog2-server	14	187.6	25.8	13.8	284.0	103.5	106.3	1.8	6.0	2.6	89.6
apache/commons-pool	16	20.0	6.7	33.4	303.1	56.8	57.0	0.0	96.1	0.3	3.5
openmrs/OpenMrs	20	244.1	101.7	41.7	315.0	48.1	49.8	1.8	83.6	3.5	11.1
aws/aws-sdk-java	7	134.1	58.0	43.5	424.0	96.5	97.2	0.2	45.2	0.7	54.0
jankotek/mapdb	7	173.6	81.7	47.3	449.1	67.3	68.5	0.6	77.9	1.6	19.9
Average(LONG)	18.4	155.9	**54.1**	**40.5**	236.8	**68.2**	**69.7**	0.8	70.8	2.3	26.2
Average(OVERALL)	26.2	91.7	**29.4**	**35.2**	93.0	**81.0**	**87.6**	0.9	41.3	6.5	51.3

run the impacted tests, and update dependencies for the next run, while the RetestAll time is the time to compile the application and run all the tests. We include compile time in this evaluation because when a change occurs, a continuous integration system that peforms the testing, such as Travis [20], typically also compiles the application as well, so we wanted to evaluate any savings in the overall build time when using STARTS. Table I shows for each project (sorted by increasing RetestAll time), the number of revisions over which we evaluated STARTS (SHAs [#]), the average number of all tests across all those revisions (ALL), the average number of tests selected by STARTS (Selected [#]), the average percentage of all tests selected by STARTS (Selected [%]), RetestAll time (RTA[s]), and the average percentage of RetestAll time that STARTS takes, shown for both the "online" mode (Online Time [%]) (Section III-B) that includes time for the *a*, *e*, and *g* phases, and the "offline" mode (Offline Time [%]) that does not include the time for the *g* phase. The last columns show a breakdown of the STARTS time into *a*, *e*, *g* and compilation (Comp.) times. In the offline mode, a developer can get test results faster by not having to wait until STARTS finishes the computation of dependencies before seeing those test results—dependency and transitive closure computation can be removed from the developer's critical path.

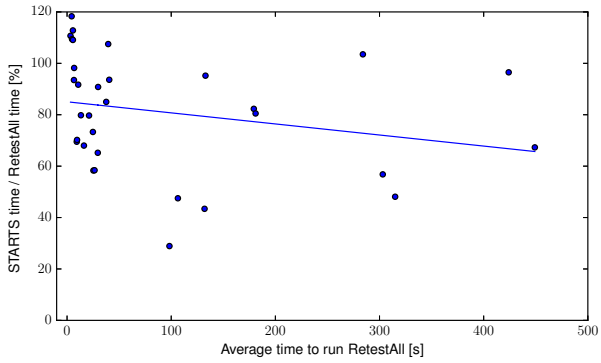

Fig. 2: Correlation between project end-to-end test time vs. percentage of time to run STARTS

We divide the projects in Table I into *short-running* if RetestAll takes less than one minute (upper part), and *long-running* if RetestAll takes more than one minute (lower part). Table I shows that STARTS runs fewer tests compared with RetestAll: STARTS selects between 4.4% (apache/commons-collections) and 84.7% (brettwooldridge/HikariCP) of all tests, with an average of 35.2% of all tests across all projects.

Table I shows that STARTS also provides time savings, with an average end-to-end time of 81.0% of RetestAll time in the offline mode, and 87.6% of RetestAll time in the online mode. STARTS provides greater time savings for long-running projects (68.2% in the offline mode and 69.7% in the online mode) than for short-running projects (87.8% in the offline mode and 96.9% in the online mode). STARTS is *more* expensive than RetestAll (i.e., the offline percentage of RetestAll time is geater than 100%) in six (of 21) short-running projects, and only one (of 11) long-running project. As expected, STARTS is better suited for long-running projects. Figure 2 plots the correlation between the average RetestAll time per project (x-axis) and the percentage time savings from the STARTS offline mode (y-axis); the Kendall-τ_b value is -0.3, and $p < 0.01$, a weak negative correlation. Finally, the breakdown of the end-to-end time shows that STARTS spends most of its non-compilation time in the *e* phase (41.3% of end-to-end time, on average), while *a* and *g* take up much smaller percentages. The time for short-running projects is dominated by compilation and these projects likely cannot benefit much from any RTS, including STARTS

V. Limitations and Future Work

We discuss some limitations of STARTS as well as future research and development directions.

Limitations of STARTS: In our previous study [8], we found that SRTS performed comparably with dynamic RTS (we evaluated against Ekstazi) in terms of time. However, we also found that SRTS is as expected, less precise than dynamic RTS and can be unsafe. (An RTS technique is *precise* if it selects to run *only* the impacted tests, and *safe* if it does not miss to select an impacted test.) We also found that reflection was the only cause of unsafety of SRTS when compared with Ekstazi. STARTS does not yet address these safety and precision limitations of SRTS. STARTS can be unsafe when the path between tests and changed types can only be reached via reflection, and is inherently imprecise because the static dependencies it finds among the types in the application may not be runtime dependencies. STARTS also assumes that there is no test-order dependence [21], [22].

Future Work: Open research directions are how to make SRTS more precise, how to make SRTS safer with respect to other potential sources of unsafety (such as dependency of tests on external files or native code), and how to apply SRTS to other programming languages. Future development plans include (i) developing faster checksum and dependency storage formats, (ii) supporting other build systems , such as Bazel or Gradle, (iii) making STARTS usable in continuous-integration systems, e.g., Jenkins or Travis, and (iv) evaluating STARTS on even larger applications than those we have evaluated so far and further improving the scalability of STARTS.

VI. Conclusion

We presented STARTS, a publicly available, purely static, class-level RTS tool. STARTS is motivated by the need in the research community to further investigate static RTS approaches, because its counterpart, dynamic RTS, is gaining some adoption in practice. We discussed the firewall technique on which STARTS is based, and we presented the usage, design, and implementation of STARTS. Our evaluation on 32 open-source projects showed that STARTS can save time compared to RetestAll, and we highlighted some future research and development directions. We believe that STARTS can help to facilitate collaboration and contribute greatly to further (static) RTS research.

Acknowlegements

We thank Felicia Chandra, Alex Gyori, Milica Hadzi-Tanovic, Farah Hariri, Hanjie Wang, Xin Wei, Lingming Zhang, and Peiyuan Zhao for their contributions to STARTS. Deniz Arsan and David Craig provided valuable feedback on a draft of this paper. This work was partially supported by National Science Foundation grants CCF-1409423 and CCF-1421503. We gratefully acknowledge support for regression testing from Google, Microsoft, and Qualcomm.

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *STVR*, vol. 22, no. 2, 2012.
[2] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *FSE*, 2014.
[3] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[4] N. York, "Tools for continuous integration at Google scale," Jan 2011, https://www.youtube.com/watch?v=b52aXZ2yi08.
[5] 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.
[6] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002.
[7] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE*, 2015.
[8] 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.
[9] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015.
[10] ——, "Ekstazi: Lightweight test selection," in *ICSE Demo*, 2015.
[11] "Yet Another Simple Graph Library," https://github.com/TestingResearchIllinois/yasgl.
[12] "JGraphT," http://jgrapht.org/.
[13] "Introduction to Maven 2.0 Plugin Development," https://maven.apache.org/guides/introduction/introduction-to-plugins.html.
[14] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *ICSM*, 1990.
[15] 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.
[16] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *FSE*, 2004.
[17] "jdeps," https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html.
[18] "Maven is broken by design," https://blog.ltgt.net/maven-is-broken-by-design/.
[19] "Guide to naming conventions," https://maven.apache.org/guides/mini/guide-naming-conventions.html.
[20] "Travis-CI," https://travis-ci.org/.
[21] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: detecting state-polluting tests to prevent test dependency," in *ISSTA*, 2015.
[22] S. Zhang, D. Jalali, J. Wuttke, K. Mucslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.