Towards Refactoring-Aware Regression Test Selection

Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim*, Don Batory, and Milos Gligoric
The University of Texas at Austin, *Iona College
{kaiyuanw,cgzhu,ahmetcelik}@utexas.edu,jkim@iona.edu,batory@cs.utexas.edu,gligoric@utexas.edu

ABSTRACT

Regression testing checks that recent project changes do not break previously working functionality. Although important, regression testing is costly when changes are frequent. Regression test selection (RTS) optimizes regression testing by running only tests whose results might be affected by a change. Traditionally, RTS collects dependencies (e.g., on files) for each test and skips the tests, at a new project revision, whose dependencies did not change. Existing RTS techniques do not differentiate behavior-preserving transformations (i.e., refactorings) from other code changes. As a result, tests are run more frequently than necessary.

We present the first step towards a refactoring-aware RTS technique, dubbed Reks, which skips tests affected only by behavior-preserving changes. Reks defines rules to update the test dependencies without running the tests. To ensure that Reks does not hide any bug introduced by the refactoring engines, we integrate Reks only in the pre-submit testing phase, which happens on the developers' machines. We evaluate Reks by measuring the savings in the testing effort. Specifically, we reproduce 100 refactoring tasks performed by developers of 37 projects on GitHub. Our results show that Reks would not run, on average, 33% of available tests (that would be run by a refactoring-unaware RTS technique). Additionally, we systematically run 27 refactoring types on ten projects. The results, based on 74,160 refactoring tasks, show that Reks would not run, on average, 16% of tests (max: 97% and SD: 24%). Finally, our results show that the Reks update rules are efficient.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution*;

KEYWORDS

Regression test selection, behavior-preserving changes, Reks

ACM Reference Format:
Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim*, Don Batory, and Milos Gligoric. 2018. Towards Refactoring-Aware Regression Test Selection. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180254

1 INTRODUCTION

Regression testing runs available tests against each project revision to check that recent changes did not break previously working functionality. Although important, regression testing is costly due to both the number of tests and the number of revisions, as recently reported by large software organizations, including Google and Microsoft [12, 30, 31, 70, 74].

Regression test selection (RTS) [21, 60, 79] techniques optimize regression testing by skipping tests that are unaffected by recent project changes. Traditionally, an RTS technique collects *dependencies* (e.g., statements, methods, or files) for each test and then runs tests whose dependencies are modified. RTS is considered (1) *safe* if it guarantees to select all affected tests (under standard assumptions [60], e.g., there are no changes in the environment between test runs [25]), and (2) *precise* if it does not select unaffected tests.

Many RTS techniques have been proposed over the last several decades [6, 21–23, 28, 44, 49, 54, 55, 59, 60, 68, 70, 77–80]; these techniques differ in granularity on which they collect dependencies. For example, Ekstazi [3, 19, 28] is a recent RTS technique that collects dependencies on *files*, i.e., Ekstazi collects files that are used by each test class and runs a test class (at the new project revision) if any of its dependent files changes.

Motivation. Despite recent improvements of RTS techniques, no RTS technique specially treats *behavior-preserving changes*, i.e., *refactorings* [24, 51, 52, 73][1], which are common in practice [63]. Our key insight is that formally proven behavior-preserving changes [7, 14, 62] do *not* impact the test outcomes, and therefore no test needs to be run after refactorings. However, the existing RTS techniques run all tests affected by refactorings, e.g., if a developer renames a method in class $\hat{c}$, and the method is used in class $\hat{c}'$, Ekstazi selects all tests that depend on either $\hat{c}$ or $\hat{c}'$. In other words, the existing RTS techniques are imprecise (i.e., too conservative) for changes made by refactorings.

Technique. We present the first step towards a *refactoring-aware* RTS technique, dubbed Reks, which does not run tests that are affected only by refactorings. At the same time, because refactorings modify the structure of code, dependencies for tests need to be updated, e.g., when a class is renamed (from $\hat{c}$ to $\hat{c}'$), each test that depended on the old class $\hat{c}$ has to depend on the new class $\hat{c}'$. Reks defines necessary *rules to update the set of dependencies* of each test, after refactorings are performed, *without running the tests*. These rules require a close integration of refactoring engines and RTS tools. Whenever a user performs an *automated* refactoring, the refactoring engine should notify Reks about the files that are affected by the refactoring and the way in which those files are affected. Reks optimizes regression testing even in cases when *changes are made by a mix of refactorings and (manual) non-refactoring changes*.

Implementation. We implemented a prototype of the Reks rules as a library, which can be easily integrated into any IDE. We further integrated the Reks library into Eclipse [17] via a plugin.

[1] We consider traditional refactoring types that are frequently integrated in IDEs, e.g., rename method, extract method, etc.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180254

	pre-submit testing	post-submit testing
Executed on	local machine	continuous integration system
Uses	RTS + Reks	test-case prioritization
Runs	selected tests	all tests

Figure 1: Recent work, which involved Google developers, described a testing process that includes two phases: pre-submit and post-submit [20]. Pre-submit testing executes only selected tests locally, while post-submit testing executes *all tests* on a continuous integration system. Reks integrates into the pre-submit phase to skip tests affected only by refactoring changes; the integration into the pre-submit phase ensures that potential bugs introduced by the refactoring tools will be detected in the post-submit phase.

To ensure that Reks does not hide any bug introduced by the refactoring engines [16, 27, 46, 67], we integrate Reks only in the *pre-submit testing* phase, which happens on the developers' machines [29]. Large companies, including Google, split testing into the pre-submit and post-submit phases to balance the testing load [20]; many other developers follow the same approach [29]. If any bug remains undetected in the pre-submit phase, the bug would be detected in the post-submit phase that *runs all available tests*. Figure 1 shows the way in which Reks should be integrated into the testing process.

Evaluation. We evaluated Reks by measuring savings in the testing effort. Specifically, we reproduced 100 refactoring tasks performed by open-source developers of 37 projects on GitHub [63]. Our results show that Reks would not run, on average, 33% (max: 100% and SD: 33%) of *available* tests; these tests would be run by a refactoring-unaware RTS. Additionally, we systematically ran 27 refactoring types on ten open-source projects and measured (1) the percent of tests not run due to Reks, and (2) time to update the set of dependencies with the Reks rules. The results, based on 74,160 refactoring tasks, show that Reks does not run, on average, 16% of available tests (max: 97% and SD: 24%). Our experiments also revealed that only 0.47% test methods fail due to bugs in refactoring engines. Although these bugs would be hidden in the pre-submit testing phase, the bugs would be detected in the post-submit testing phase (see Figure 1). Finally, our experiments show that the update rules are efficient regardless of the size of the target project.

2 MOTIVATING EXAMPLE

This section illustrates an existing RTS technique, shows the limitations of the technique (and other existing RTS techniques), and introduces a novel refactoring-aware RTS technique, dubbed Reks.

Project. Consider the `Byte-Buddy` project [11], which is a popular light-weight Java library that supports runtime code generation and manipulation; the project is hosted on GitHub. At revision `35da279`, the `Byte-Buddy` project had 3,150 test methods in 362 test classes that execute in about 37 seconds. Developers then made several changes and obtained a new project revision (`f1dfb66`). Figures 2a and 2b show code before and after the change, respectively. The change triggered the execution of all (362) available test classes (https://travis-ci.org/raphw/byte-buddy).

Ekstazi. Suppose that the `Byte-Buddy` project uses the Ekstazi tool [3, 19, 28]. Ekstazi collects for each test class the set of dynamically accessed files. The set of collected files includes both the executable (e.g., Java classfiles) and non-executable (e.g., property) files.

During the execution of the tests at old revision (`35da279`), Ekstazi would collect a set of dependencies for each test class. Figure 3a shows a subset of dependencies for three test classes. Each dependency is a pair of the file name and the checksum of the file.

After the developers make changes between `35da279` (old) and `f1dfb66` (new), Ekstazi analyzes the changes and runs only tests that are affected by the changes. Specifically, to detect modified files, Ekstazi computes the checksum for each file and finds the files whose checksum has changed. Any test that depends on at least one of the changed files is selected for the execution.

In our example, the developers modified `AbstractBase` and `ForLoadedExecutable` files. We can see in Figure 3 that some tests depend on the modified files and are selected. Specifically, at new revision `f1dfb66`, Ekstazi selects 1,248 test methods in 88 test classes (out of 362 test classes) taking about 27 seconds (including time to find modified files). Additionally, during the execution of the affected tests, Ekstazi collects new dependencies for these tests. Figure 3b shows the updated dependencies for our example.

Reks. What is interesting about the changes between `35da279` and `f1dfb66` is that these changes are *refactorings*, i.e., behavior preserving transformations. Specifically, by manually analyzing the change, we found that a developer pulled up [71, 72] the `wrap(List<ParameterDescription>)` method from subclasses to the superclass (`AbstractBase`) between two revisions.

As we illustrated above, Ekstazi, like other existing RTS techniques, is *refactoring-unaware*, i.e., it runs tests affected by refactorings. However, behavior-preserving changes [7, 14, 62] do *not* impact the test outcomes, and therefore no test has to be run after refactorings. (We are aware that refactorings may introduce bugs [16, 27, 46, 67], as we discussed in the introduction.)

Reks is a novel RTS technique that skips running the tests after refactorings, but updates the dependencies for each test. Reks is built on top of Ekstazi, i.e., it keeps dependencies on files and defines the rules to update such dependencies. In our example, Reks would *not* run any test between `35da279` and `f1dfb66`, but it would update the set of dependencies to match those in Figure 3b. Reks also supports mixed changes, i.e., changes with refactorings and other manual edits. The following section defines the update rules.

3 TECHNIQUE

This section formally introduces the Reks update rules, and Section 4 illustrates these rules using two refactoring types.

3.1 Preliminaries

We write P^t to denote the state of the project P at time t. P^t includes all files on disk, at time t, which belong to the project. *Project revisions* are states of the project that are observable in the version-control system (i.e., commits on GitHub). There can be one or more changes between two project revisions [29, 61].

A *test session* identifies a point in time when tests were executed; a test session is started either by a developer or a continuous integration system, e.g., Travis CI [33]. For simplicity of exposition, we assume that a test session runs all affected tests. A test session

```
abstract class AbstractBase { ...
}
class ForLoadedExecutable extends AbstractBase {
  @Override
  protected ParameterList wrap(List<ParameterDescription> values) {
    return new Explicit(values);
  } ...
} ...
```

(a) Code before refactoring

```
abstract class AbstractBase {
  @Override
  protected ParameterList wrap(List<ParameterDescription> values) {
    return new Explicit(values);
  } ...
}
class ForLoadedExecutable extends AbstractBase { ...
} ...
```

(b) Code after refactoring

Figure 2: An example refactoring change in the Byte-Buddy project. Developers used Pull Up refactoring to move the method `wrap` from sublasses (e.g., `ForLoadedExecutable`) to their superclass (`AbstractBase`).

JavaMethodTest
(ByteBuddyCommons, 4677), (JavaMethod, 1783), ...
JavaInstanceMethodTypeTest
(AbstractBase, 7263), (ForLoadedExecutable, 4267), ...
JavaInstanceMethodHandleTest
(AbstractBase, 7263), (ForLoadedExecutable, 4267), ...

(a) Dependencies for three tests at 35da279

JavaMethodTest
(ByteBuddyCommons, 4677), (JavaMethod, 1783), ...
JavaInstanceMethodTypeTest
(AbstractBase, 1076), (ForLoadedExecutable, 1291), ...
JavaInstanceMethodHandleTest
(AbstractBase, 1076), (ForLoadedExecutable, 1291), ...

(b) Dependencies for three tests at f1dfb66

Figure 3: Impact of refactoring changes from Figure 2 on Ekstazi dependencies. (a) shows test dependencies for three tests before refactoring and (b) shows test dependencies for the same tests after refactoring. Each dependency is a pair of a file and its checksum. We highlight the dependencies that are changed and tests that are affected between the two revisions.

can be triggered at any point in between revisions (i.e., pre-submit testing phase) [29]. We write P^{old} to denote a project state after the latest test session. A *refactoring task* identifies a point in time when one of the refactoring types is invoked by a developer. We write P^{middle} and P^{new} to denote the states immediately before and immediately after the latest refactoring task, respectively.

We define a *sequence* as a triple $(P^{old}, P^{middle}, P^{new})$. Figure 4 shows an example project timeline, including revisions, test sessions, a refactoring task, and a sequence.

We define a function $\underline{\text{tests}}(P^t)$ that returns a set of available tests at the state P^t. Similarly, we define a function $\underline{\text{files}}(P^t)$ that returns a set of files (i.e., normalized paths) that belong to the project at the state P^t.

We will use $\hat{c}$ as a variable of type file, and function $\underline{\text{content}}(P^t, \hat{c})$ returns the content of the file at state P^t. We also define a function $\underline{\text{cksum}}(\underline{\text{content}}(P^t, \hat{c}))$ that computes the checksum of the file's content as a string at state P^t; the function returns a special value if the file does not exist. If the state is clear from the context, we simply write $\underline{\text{cksum}}(\hat{c})$. We use $\hat{s}$ as a free variable of type string.

Further we overload the $\underline{\text{cksum}}$ function to accept a project state as an argument, i.e., $\underline{\text{cksum}}(P^t)$. This function returns a set of pairs $(\hat{c}, \underline{\text{cksum}}(\hat{c}))$ for all files at the state P^t, i.e., $\underline{\text{cksum}}(P^t) = \{(\hat{c}, \underline{\text{cksum}}(\hat{c})) \mid \hat{c} \in \underline{\text{files}}(P^t)\}$. We define a function $\underline{\text{deps}}(P^t, \tau)$, s.t., $\underline{\text{deps}}(P^t, \tau) \subseteq \underline{\text{cksum}}(P^t)$, where $\tau \in \underline{\text{tests}}(P^t)$, that returns a set of *dependencies* (i.e., a set of $(\hat{c}, \underline{\text{cksum}}(\hat{c}))$ pairs) for a given test in the given project state. The set of dependencies for each test is collected by a code coverage tool [28, 35, 64, 80]. We write $\underline{\text{deps}}(P^t, \tau) \cup (\hat{c}, \underline{\text{cksum}}(\hat{c}))$ to denote an addition of the pair $(\hat{c}, \underline{\text{cksum}}(\hat{c}))$ to the set of dependencies of τ for state P^t. Similarly, we write $\underline{\text{deps}}(P^t, \tau) \setminus (\hat{c}, \hat{s})$ to denote a removal of the pair $(\hat{c}, \hat{s})$ from the set of dependencies. Finally, we write $\underline{\text{deps}}(P^t, \tau) \uplus (\hat{c}, \underline{\text{cksum}}(\hat{c}))$ to denote an update of the set, i.e., $\underline{\text{deps}}(P^t, \tau) \uplus (\hat{c}, \underline{\text{cksum}}(\hat{c})) = \underline{\text{deps}}(P^t, \tau) \setminus (\hat{c}, \hat{s}) \cup (\hat{c}, \underline{\text{cksum}}(\hat{c}))$, where $\hat{s}$ is an old checksum of $\hat{c}$. The addition, removal, and update are overloaded to also work with a set of pairs.

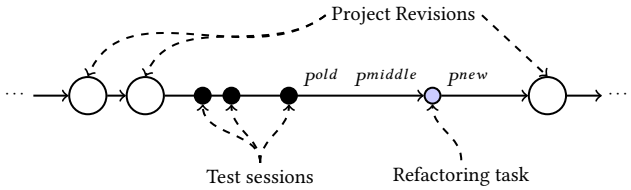

Figure 4: An example project development timeline that illustrates revisions (which are available in version control history of the project), test sessions (which are points in time when developers run tests), refactoring tasks (which are points in time when developers run one of the refactorings), and various project states.

We define a function $\underline{\text{diff}}(P^{old}, P^{new})$ that returns a set of modified files between two states, i.e., $\underline{\text{diff}}(P^{old}, P^{new}) = \{\hat{c}' \mid \hat{c} \in \underline{\text{files}}(P^{old}) \wedge \hat{c}' \in \underline{\text{files}}(P^{new}) \wedge \underline{\text{cksum}}(\hat{c}) \neq \underline{\text{cksum}}(\hat{c}') \wedge \hat{c} = \hat{c}'\}$. We also define functions that return sets of added and deleted files, i.e., $\underline{\text{added}}(P^{old}, P^{new}) = \{\hat{c} \mid \hat{c} \notin \underline{\text{files}}(P^{old}) \wedge \hat{c} \in \underline{\text{files}}(P^{new})\}$ and $\underline{\text{deleted}}(P^{old}, P^{new}) = \{\hat{c} \mid \hat{c} \in \underline{\text{files}}(P^{old}) \wedge \hat{c} \notin \underline{\text{files}}(P^{new})\}$.

Ξ^{dev} denotes files that are added, modified, and deleted by developers in a single sequence $(P^{old}, P^{middle}, P^{new})$, i.e., $\Xi^{dev} = \underline{\text{diff}}(P^{old}, P^{middle}) \cup \underline{\text{added}}(P^{old}, P^{middle}) \cup \underline{\text{deleted}}(P^{old}, P^{middle})$. Similarly, Ξ^{rft} denotes a set of files that are modified by the invoked refactoring, i.e., $\Xi^{rft} = \underline{\text{diff}}(P^{middle}, P^{new})$. Further, we write $\Pi^{\pm files}$ to denote a set of file pairs that describe replacements of files, i.e., the first file of each pair is a file that is replaced with the second file of the pair. Similarly, we write $\Pi^{\pm elems}$ to denote a set of file pairs such that some code elements (e.g., methods or fields) from the first file are moved to the second file; both files exist after the elements are moved. The last three sets (Ξ^{rft}, $\Pi^{\pm files}$ and $\Pi^{\pm elems}$) can be extracted from the refactoring engine when a refactoring task is invoked.

3.2 Update Rules

Let $(P^{old}, P^{middle}, P^{new})$ be a sequence. We define three rules to update the set of dependencies for τ at state P^{new}, where $\tau \in \underline{\text{tests}}(P^{old}) \wedge \tau \in \underline{\text{tests}}(P^{new})$.

Modified dependencies. For the set of dependencies that are modified by the refactoring engine (Ξ^{rft}), we have to recompute the checksum for each of these dependencies. However, we do not update the checksum of the dependencies that are also modified by the developers (Ξ^{dev}) in the current sequence.

$$\underline{\text{deps}}(P^{new}, \tau) = \underline{\text{deps}}(P^{old}, \tau) \uplus \{(\hat{c}, \underline{\text{cksum}}(\hat{c})) \mid \hat{c} \in \Xi^{rft} \wedge \hat{c} \notin \Xi^{dev} \wedge (\hat{c}, \hat{s}) \in \underline{\text{deps}}(P^{old}, \tau)\} \quad (1)$$

Replaced files. For each pair of files that describes a file replacement ($\Pi^{\pm files}$), we have to remove the old file from the set of dependencies and include a new file in the set of dependencies. As in the previous rule, we have to skip the files that are modified by the developers (Ξ^{dev}), prior to file replacement, in the current sequence. Therefore, we update the set of dependencies as follows:

$$\underline{\text{deps}}(P^{new}, \tau) = \underline{\text{deps}}(P^{old}, \tau) \setminus \{(\hat{c}, \hat{s}) \mid (\hat{c}, \hat{c}') \in \Pi^{\pm files} \wedge \hat{c} \notin \Xi^{dev}\} \cup \{(\hat{c}', \underline{\text{cksum}}(\hat{c}')) \mid (\hat{c}, \hat{c}') \in \Pi^{\pm files} \wedge \hat{c} \notin \Xi^{dev} \wedge (\hat{c}, \hat{s}) \in \underline{\text{deps}}(P^{old}, \tau)\} \quad (2)$$

Moved elements. For each pair of files that identifies movement of some code elements from one file to another ($\Pi^{\pm elems}$), we have to include the file into which the elements are moved to the set of dependencies. At the same time, we keep the file from which the elements are moved in the set of dependencies.

$$\underline{\text{deps}}(P^{new}, \tau) = \underline{\text{deps}}(P^{old}, \tau) \cup \{(\hat{c}', \underline{\text{cksum}}(\hat{c}')) \mid (\hat{c}, \hat{c}') \in \Pi^{\pm elems} \wedge \hat{c}' \notin \Xi^{dev} \wedge (\hat{c}, \hat{s}) \in \underline{\text{deps}}(\tau, P^{old})\} \quad (3)$$

Rules (2) and (3) are mutually exclusive, because the former requires that the first element of each file pair does not exist at P^{new}, and the latter requires that the first element of each file pair exists at P^{new}. Rule (1) and Rule (2)/Rule (3) always commute for a single sequence.

3.3 Advanced Considerations

Added and removed tests. There is no need to specially treat newly added tests (i.e., $\tau \notin \underline{\text{tests}}(P^{old}) \wedge \tau \in \underline{\text{tests}}(P^{new})$). These tests will have no associated dependencies initially, so they will always be selected by the RTS technique [28]. Deleted tests (i.e., $\tau \in \underline{\text{tests}}(P^{old}) \wedge \tau \notin \underline{\text{tests}}(P^{new})$) do not require special treatment either, because their set of dependencies will not be used in the subsequent test runs [28].

Relaxed definition of a sequence. In the update rules, we assume that each sequence starts with a test session and ends at the time of the first subsequent refactoring task. This means that there can be an arbitrary number of manual changes between a test session and a refactoring task, but there has to be at least one test session between two refactoring tasks. We set this requirement only to simplify the definition of $\underline{\text{d}}$iff, $\underline{\text{a}}$dded, and $\underline{\text{d}}$eleted functions; there is no such limitation in our implementation. This requirement is relaxed by computing the modified, added, and deleted files at every refactoring invocation by comparing the file checksum with either the checksum at the previous refactoring task or previous test session (whichever one is the latest in the project development history) and then taking the union with already existing sets of modified, added, and deleted files.

Overapproximation. Rule (3) may lead to an overapproximation of the set of dependencies. Namely, if τ used *only* the moved elements in the original file, then τ does not have to depend on the original file after the elements are moved. However, we are unable to identify if τ should not depend on the original file any more, because we keep the set of dependencies on files, and we do not know the reason why τ depended on that file in the first place. This is not a problem however, because overapproximation of the set of dependencies does not impact the *safety* of regression test selection [28].

We can reduce the overapproximation by reasoning about each refactoring type independently. For example, consider the Move Method refactoring that moves an *instance method* from file $\hat{c}$ to $\hat{c}'$. Based on our Rule 3, we would always add $\hat{c}'$ in the set of dependencies. However, based on the definition of Move Method refactoring (at least in the Eclipse JDT), there is no way that the test could start depending on $\hat{c}'$ in this case. Although case analysis could be interesting, it would lead to a less elegant solution and it would be specific to a refactoring engine.

4 UPDATE RULES ILLUSTRATED

This section illustrates the Reks rules using the Move Method and Convert Anonymous to Nested refactoring types that are available in the Eclipse JDT [36]. We chose to present Move Method because it is one of the most frequently used refactorings in practice [48]. We chose Convert Anonymous to Nested because it creates a new file and it has significant impact on regression testing (Section 5). In this section, with the goal to simplify the presentation, we assume that a developer runs a test session prior to each refactoring task, i.e., $P^{old} = P^{middle}$.

4.1 Move Method

Move Method refactoring moves the selected ("target") method from file $\hat{c}$ to $\hat{c}'$ and updates all references to this method. A non-static (i.e., "instance") method can be moved via a parameter or a field using a JDT refactoring [36]. Moving an instance method "via parameter" means that we choose one of the parameter types as the destination of the method. Moving a method "via field" means that we choose one of the fields and use its type as the destination of our method.

Test depends on the declaring file. Consider a test class that depends on file $\hat{c}$ and the target method is moved from $\hat{c}$ to $\hat{c}'$. An example of such code is shown in Figure 5a. We invoke a refactoring task on method `S.m` and choose `D` (which is the type of the parameter) as the destination type. The result of the refactoring is shown in Figure 5b. The execution of the refactoring task will collect the following information:

$\Xi^{rft} = \{S, D\}$	Set of modified files
$\Pi^{\pm files} = \emptyset$	Set of pairs that describe replaced files
$\Pi^{\pm elems} = \{(S, D)\}$	Set of pairs that described moved elements

Looking at the rules (1), (2), and (3), we can see that the Rule (2) is not applicable in this case, because the set of pairs that describes the

```
class T {
   void t() {
      new S().a();
   }
}
class S {
   void m(D d) {}
   void a() {}
}
class D {}
```

(a) Before: $deps(P^{old}, T) = \{(S, 52749)\}$

```
class T {
   void t() {
      new S().a();
   }
}
class S {
   void a() {}
}
class D {
   void m() {}
}
```

(b) After moving m: $deps(P^{new}, T) = \{(S, 93752), (D, 88792)\}$

Figure 5: Move instance method (via parameter) when a test class depends on the class that declares the moved method.

```
class T {
   void t() {
      D.a();
   }
}
class S {
   void m(D d) {}
}
class D {
   static void a() {}
}
```

(a) Before: $\underline{deps}(P^{old}, T) = (D, 38562)$

```
class T {
   void t() {
      D.a();
   }
}
class S {}
class D {
   static void a() {}
   void m() {}
}
```

(b) After moving m: $\underline{deps}(P^{new}, T) = (D, 24759)$

Figure 6: Move instance method (via parameter) when a test class depends on the destination class.

replaced files is empty. (In fact, Move Method refactoring never uses Rule (2), because this refactoring cannot introduce a new file.)

By applying the Rule (1), we update the checksum of the file where the moved method was declared previously. This is necessary, because some code was changed and the checksum is not the same any longer. Specifically, we have the following:

$$\underline{deps}(P^{new}, T) = \underline{deps}(P^{old}, T) \uplus \{(S, 93752)\}$$
$$= \{(S, 52749)\} \uplus \{(S, 93752)\} = \{(S, 93752)\}$$

Further, by applying Rule (3), we include the additional file (i.e., destination of the moved method) in the set of dependencies.

$$\underline{deps}(P^{new}, T) = \{(S, 93752)\} \cup \{(D, 88792)\}$$

As discussed in Section 3.3, Move Method refactoring is one of few that may overapproximate the set of dependencies and this can be fixed by refining the rules for such refactorings with case analysis. For example, in this particular example, we can see that test class T does not actually use D after refactoring. In fact, if T was using S.m before the refactoring, it would have already been dependent on D.

Test depends on the destination file. Consider a test class that depends on the destination file $\hat{c}'$ of a Move Method refactoring task. An example of such code is shown in Figure 6. We invoke refactoring on S.m and choose D as the destination. As before, the

```
class T {
   void t() {
      new R().a();
   }
}
class S {
   void m(D d) {}
}
class D {}
class R {
   void a() {}
   void b() {
      new S().m(new D());
   }
}
```

(a) Before: $\underline{deps}(P^{old}, T) = (R, 63958)$

```
class T {
   void t() {
      new R().a();
   }
}
class S {}
class D {
   void m() {}
}
class R {
   void a() {}
   void b() {
      new D().m();
   }
}
```

(b) After moving m: $\underline{deps}(P^{new}, T) = (R, 83475)$

Figure 7: Move instance method (via parameter) when a test class depends on the class that references the method.

```
class T {
   void t() {
      new R().m();
   }
}
class R {
   void m() {
   new P(){};
   }
}
class P {}
```

(a) Before: $\underline{deps}(P^{old}, T) = \{(R, 52749), (P, 52749), (P\$1, 52749)\}$

```
class T {
   void t() {
      new R().m();
   }
}
class R {
   static class D extends P {}
   void m() {
      new D();
   }
}
class P {}
```

(b) After: $\underline{deps}(P^{new}, T) = \{(R, 22334), (P, 52749), (D, 72892)\}$

Figure 8: Convert anonymous class to nested class when a test class depends on the anonymous class.

execution of the refactoring will collect the following information:

$\Xi^{rft} = \{S, D\}$	Set of modified files
$\Pi^{\pm files} = \emptyset$	Set of pairs that describe replaced files
$\Pi^{\pm elems} = \{(S, D)\}$	Set of pairs that described moved elements

It is necessary to apply Rule (1) because the checksum of the destination file changed when the new method was added. We have:

$$\underline{deps}(P^{new}, T) = \underline{deps}(P^{old}, T) \uplus \{(D, 24759)\}$$
$$= \{(D, 38562)\} \uplus \{(D, 24759)\} = \{(D, 24759)\}$$

We can see that the Rule (3) is not applicable to our test T, because the source file (S) is not in the original set of dependencies for T, i.e., $(S, \underline{cksum}(S)) \notin \underline{deps}(P^{old}, T)$.

Test depends on a reference file. Consider a test class that depends on a file that references the moved method. An example of such code is shown in Figure 7. Test T is affected by the change because the reference file (R) is modified when the method is moved from S to D. The reasoning about the update rules is the same as in the previous case when test was dependent on D, so we do not discuss this case in more detail.

4.2 Convert Anonymous Class to Nested

Convert Anonymous to Nested refactoring type converts an anonymous class $\hat{c}$ to a new nested class $\hat{c}'$. This refactoring requires two parameters: a name of the new class and a flag if the new class should be an inner or a static nested class.

It is important to observe that although anonymous classes and nested classes are in the same *source file* as their enclosing class, they are in *different classfiles* from their enclosing class.

Consider, as shown in Figure 8, a test class `T` that depends on `R`, `P`, and `P$1`; `P$1` refers to the anonymous class. (A test that depends on an anonymous class always depends on the enclosing class too, because the anonymous class can only be referenced from its enclosing class.) We invoke refactoring on `new P()` and choose `D` as the name for the new nested class. The execution of the refactoring task will collect the following information:

$\Xi^{rft} = \{R\}$	Set of modified files
$\Pi^{\pm files} = \{(P\$1, D)\}$	Set of pairs that describe replaced files
$\Pi^{\pm elems} = \emptyset$	Set of pairs that described moved elements

It is necessary to apply the Rule (1), because the checksum of the enclosing class (`R`) always changes.

$$\underline{\text{d}}\text{eps}(P^{new}, T) = \{(R, 52749), (P, 52749), (P\$1, 52749)\} \uplus \{(R, 22334)\}$$
$$= \{(R, 22334), (P, 52749), (P\$1, 52749)\}$$

We also have to apply the Rule (2), as one of the files was replaced:

$$\underline{\text{d}}\text{eps}(P^{new}, T) = \{(R, 22334), (P, 52749), (P\$1, 52749)\}$$
$$\setminus\{(P\$1, 52749)\} \cup \{(D, 72892)\} = \{(R, 22334), (P, 52749), (D, 72892)\}$$

5 EVALUATION

We assess the benefits of the Reks technique by answering the following research question:

RQ1: How many tests would have been skipped by Reks had it been used by open-source developers?

Furthermore, we evaluated the benefits of Reks by systematically performing refactorings (one at a time) on open-source projects and answering the following questions:

RQ2: How many tests does Reks skip on average if refactorings are systematically performed?

RQ3: How many tests does Reks skip on average for various refactoring types if refactorings are systematically performed?

RQ4: What is the cost of the Reks update rules and how does this cost compare to the test execution time?

Additionally, we collect the data to study the frequency of bugs (i.e., non-behavior preserving transformations) introduced by refactoring engines. As we mentioned earlier (see Section 1), Reks should be integrated in the pre-submit testing phase, so that any potential bug is discovered in the post-submit phase, which executes all available tests. Specifically, we answer the following question:

RQ5: How many test methods fail, on average, due to refactoring tasks (performed by the Eclipse refactoring engine)?

5.1 Refactorings in Open-Source Projects

5.1.1 Methodology. We evaluated Reks in a realistic setting with *pure refactorings* (i.e., only refactoring changes) and *mixed changes* (i.e., refactorings + non-refactoring changes) performed by open-source developers. We do not include results for only non-refactoring changes, because Reks always gives the same results as Ekstazi in those cases [28].

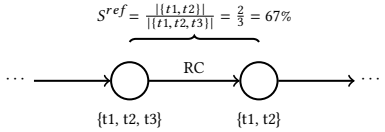

Figure 9: An example to illustrate the way we compute S^{ref} for a pure refactoring change (RC); the figure shows tests run by Ekstazi, and Reks always runs zero tests for RC.

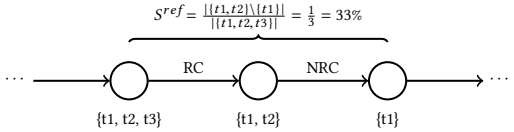

Figure 10: An example to illustrate the way we compute S^{ref} for mixed changes; we split each mixed changes to refactoring (RC) and non-refactoring (NRC) changes only to enable the experiment; the figure shows tests run by Ekstazi.

To find pure refactorings and mixed changes, we followed two paths. First, we used projects and revisions detected in a recent study that analyzed refactorings on GitHub [63]. Second, we searched on GitHub for revisions with the commit messages that include either "refactor", "rename", or "move" in Java repositories of two organizations: Apache and Google.

We executed the following steps for each discovered pair of (*Project*, *Revision*) with the goal to select the pairs for our study:

a) Clone the *Project* and checkout the *Revision*.
b) If the *Project* does not use Maven, which is currently supported by Reks, discard the (*Project*, *Revision*) pair.
c) Build the *Project* (for the *Revision* and its parent revision). If the build is not successful, discard the pair. We had to discard a large number of pairs due to the broken builds. However, this was not surprising [75].
d) Manually confirm (by reading the diff) that *Revision* is a pure refactoring or a mixed change.

Pure refactorings. To evaluate Reks on pure refactorings, we (1) checkout the parent revision of the *Revision*, (2) execute test classes with Reks (to collect dependencies for each test class), (3) replay the refactoring from the *Revision*, and (4) execute test classes with Reks (which always executes *zero* tests in this case but updates dependencies). We also collect, at *Revision*, the set of all available tests ($T^{available}$) and the set of test classes that are executed with Ekstazi, i.e., refactoring-unaware technique (T^{ref}). We compute the percentage of test classes that are not run due to Reks as $S^{ref} = |T^{ref}| \,/\, |T^{available}| \times 100$. We illustrate this experiment for one pure refactoring in Figure 9.

Mixed changes. To evaluate Reks on mixed changes we (1) manually split each mixed change into refactoring and non-refactoring changes (the splitting is needed only for the evaluation, but not for the actual use of the tool), (2) execute test classes with Reks (to collect dependencies), (3) replay only refactoring changes, (4) execute test classes with Reks (which executes *zero* tests, but updates

Figure 11: Distribution of S^{ref} for refactorings performed by developers of open-source projects found on GitHub; the red line shows the median and the star shows the average.

```
1  discover_affected_tests(refactoring, project):
2      run_tests_with_reks_without_rules(project)
3      elements = find_code_elements(refactoring, project)
4      for el in elements:
5          refactoring_tasks = create_tasks(refactoring, project, el)
6          for task in refactoring_tasks:
7              configure_properties(task)
8              try:
9                  if check_preconditions(task):
10                     refactored_project = perform(task)
11                 else:
12                     continue
13             except exc:
14                 continue
15             if is_successful(compile(refactored project)):
16                 affected_tests = analyze(refactored_project)
17                 store_affected_tests(affected_tests, task)
18             undo(task)
```

Figure 12: Procedure for systematically discovering affected test classes for one given refactoring and project.

the dependencies and records the set of tests selected by Ekstazi: T^{ref}), (5) replay non-refactoring changes, (6) execute test classes with Reks to record the available ($T^{available}$) and selected test classes ($T^{non-ref}$). (The order in which we replay refactoring and non-refactoring changes does not affect the outcome of our experiments.) We compute the percentage of test classes that are not run due to Reks as $S^{ref} = |T^{ref} \setminus T^{non-ref}| \,/\, |T^{available}| \times 100$. We illustrate this experiment for one mixed change in Figure 10.

5.1.2 ***RQ1****: How many tests would have been skipped by Reks had it been used by open-source developers?* Following our methodology, we performed our experiment on 100 pairs (17 are from prior study of refactorings [63] and 8% are mixed changes). Similar to recent work [63], due to the space constraints, we do not include the info for each (project, revision) pair in this paper; the list is available at http://cozy.ece.utexas.edu/reks.

Boxplot in Figure 11 shows the distribution of S^{ref} values for all 100 pairs. In sum, we find that S^{ref} for all pairs is 33% on average (min: 0%; max: 100%). We also find that S^{ref} for pure refactorings is 34% on average (min: 0%; max: 100%). Finally, S^{ref} for mixed changes is 11% on average (min: 0%; max: 64%).

5.2 Systematically Performed Refactorings

5.2.1 Methodology. We follow a systematic methodology to evaluate the impact of refactorings on RTS (**RQ2**, **RQ3**, **RQ4**, and **RQ5**) [27]. Namely, for a given project, we perform refactorings to all applicable program elements (e.g., methods), up to 5 per file, and measure the impact of each refactoring type on regression testing.

Figure 12 shows the procedure to discover affected tests for one given refactoring type and one project. (We modify a procedure presented in an earlier work used for testing refactoring engines [27].) The procedure takes two inputs: the refactoring type to perform and a project on which refactoring type will be performed.

In the first step (line 2), we run available test classes ($T^{available}$) for the given project with Reks. This step collects dependencies for all test classes, as described in Section 2 (i.e., $\underline{d}eps(P^{middle}, \tau)$ where $\tau \in \underline{t}ests(P^{middle})$). The collected dependencies are stored on disk in a Reks specific directory, which is used later in the procedure. In the second step (line 3), we find code elements for the project on which the given refactoring type can be invoked and create for each code element (line 5) a set of refactoring tasks. For example, for Move Method, the set of code elements contains all methods in the project, and the set of tasks includes moving each method to another class. In the third step (line 6), for each refactoring task, the procedure configures the refactoring task parameters (line 7); we discuss the details of the task configuration in Section 5.2.3. In the fourth step (line 9), we check if the refactoring task can be applied. Each refactoring has preconditions that need to be satisfied before a refactoring task can be performed. For JDT's Move Method, one of the preconditions is not satisfied if the target class already contains a method with the same name and same number of arguments as the method being moved. If all precondition checks succeed, we perform the refactoring task (line 10) and obtain the refactored project (i.e., P^{new}). We additionally check (line 15) that the refactored project compiles, which may not be the case if the refactoring engine encounters a bug.

In the final step (line 16), we analyze code with Reks to discover affected tests (T^{ref}); our goal is only to compute the percentage of tests that are not run by Reks but are selected with Ekstazi ($S^{ref} = |T^{ref}| \,/\, |T^{available}| \times 100$). Before the analysis phase is invoked, we need to compile the refactored project because Reks detects modified files by analyzing classfiles (rather than source files). Then Reks uses dependencies, which were saved on line 2, to find what tests are (not) affected by the refactoring task. On line 17 we store the results of Reks analysis, which we post-process to compute S^{ref}. Finally, we undo (using `git clean -xfd; git reset --hard`) the effects of the latest refactoring task (line 18), to prepare for the next iteration of the loop.

5.2.2 Projects. We briefly describe projects under study; these are the projects that we pass (one at a time) as the second argument to our procedure in Figure 12. Table 1 shows the list of the projects used in our study; we chose the projects that were used frequently in recent studies on regression testing. For each project, we show a name; a short description; repository URL from which we cloned the project; and the latest revision (SHA) of the project at the time of our study. Additionally, Table 1 shows the number of lines of code (LOC) measured with sloccount [65]; the number of test classes; and instruction, branch, and class coverages obtained with JaCoCo [35]. The last two rows show the total ($\sum$) and average (Avg) numbers for the LOC, test classes, and code coverage.

5.2.3 Refactoring Setup. Although various tools/IDEs may offer different refactorings [17, 34, 50, 53], we focus on 27 (i.e., all available) refactorings of the Eclipse JDT [17, 37]. We selected Eclipse because it is one of the most widely used tools and offers a rich set of refactorings. We do not include a description of each refactoring in this paper, but many resources are available in the existing literature [24, 36, 58].

Table 1: Projects Used in the Evaluation with Systematically Performed Refactorings.

Project	Description	URL	SHA	LOC	#Test classes	Coverage instr.	branch	class
Coll	Extension of Java Collections	apache/commons-collections	c87eeaa4	60251	159	83	77	95
Config	Generic Configuration Interface	apache/commons-configuration	8dddebf1	64341	163	87	83	98
DBCP	DB Connection Support	apache/commons-dbcp	784fb496	20547	30	45	56	95
IO	Library to Assist IO	apache/commons-io	9990c666	29159	100	86	82	100
JClassmate	Introspecting Generic Type Info	FasterXML/java-classmate	ef2fb7cd	6797	34	94	89	100
JObjectDiff	Diff and Merge Java Objects	SQiShER/java-object-diff	751574b8	9976	61	89	84	94
Lang	Manipulation of Java Classes	apache/commons-lang	17a6d163	69014	134	94	89	100
Net	Basic Internet Protocols	apache/commons-net	4450add7	26928	42	32	26	38
Pebble	Templating Engine Inspired by Twig	PebbleTemplates/pebble	f8e2e7b0	13375	30	86	79	96
Stateless4J	Lightweight Java State Machine	oxo42/stateless4j	4f12a5cb	1702	9	53	43	64
$\sum$	N/A	N/A	N/A	302090	761	N/A	N/A	N/A
Avg	N/A	N/A	N/A	30209.0	76.1	74.9	70.8	88.5

Each refactoring type accepts a set of (property, value) pairs that define the configuration of the refactoring task. For example, for Extract Method the parameters include: (1) new name, which specifies the name of the newly created method, (2) visibility, which specifies the access modifier of the newly created method, and (3) replace duplicate, which replaces duplicate code in the same class with the invocation of the newly created method. The values used in our study closely follow prior work on systematic testing of refactoring engines [27].

5.2.4 Implementation. We implemented the procedure from Figure 12 as an Eclipse plugin [18] that supports all refactorings available in Eclipse JDT. We build on top of an existing plugin, called RTR [27]. The key changes are related to Reks invocations at appropriate places and storing the results of Reks analysis. We invoke Reks as a separate Java process using `ProcessBuilder`. As the result of each Reks analysis invocation, we save three separate outputs: (1) build output (recall that we need to compile the project before we invoke Reks), (2) refactoring task details, and (3) the list of affected tests as reported by Reks analysis. We save the output of the build to check if the compilation and build were successful. We save the refactoring task details to inspect the outputs and confirm the correctness of our implementation.

Table 2: Percent of Test Classes Not Run Due to Reks (S^{ref}) Per Project for Systematically Performed Refactoring Changes.

Project	S^{ref} Max	Med	Avg	STD
Coll	48.43	1.26	3.29	6.45
Config	72.39	4.29	13.75	18.02
DBCP	89.66	20.69	26.57	22.29
IO	63.00	9.00	10.88	10.07
JClassmate	85.29	32.35	37.07	25.51
JObjectDiff	85.25	8.20	21.37	23.48
Lang	52.24	1.49	4.59	7.09
Net	38.10	2.38	4.25	7.37
Pebble	96.67	96.67	67.30	40.40
Stateless4J	77.78	11.11	22.81	19.89
Max	96.67	96.67	67.30	40.40
Avg	70.88	N/A	21.19	N/A

We apply refactoring tasks only on non-test classes to properly measure the impact of refactorings. Note that dependencies on test classes are rare [28], which means that applying refactoring tasks on a test class would modify only that single class.

We have checked the correctness of the implementation, on a number of examples, by comparing the sets of dependencies after applying the Reks update rules and sets of dependencies collected by Ekstazi. We found that the Reks rules give the expected results.

5.2.5 Collected Numbers. We ran experiments on an Intel Xeon CPU @ 2.60GHz with 16GB of RAM, running Ubuntu 16.04LTS.

Following our procedure in Figure 12, for each project and refactoring pair, we collected the number of refactoring tasks, time to execute the procedure, and average S^{ref}. In sum, we ran a total of 74,160 refactoring tasks in over 964 hours. We observed that a few refactoring tasks led to exceptions or compilation errors (~5%), which is slightly higher than reported in prior studies on testing refactorings [27]. Note that these cases (i.e., bugs in the refactoring engine that lead to non-compilable code) do not impact safety of our technique, because *a user would not run the tests in a project that does not compile*. In Section 5.2.9, we discuss bugs that change the program behavior.

5.2.6 ***RQ2****: How many tests does Reks skip on average if refactorings are systematically performed?* Table 2 shows the max, median, average, and standard deviation of S^{ref} per project (measured across all refactoring types). For example, we can see that max, median, and average S^{ref} for `DBCP` are 90%, 21%, and 27%, respectively. The average across all refactoring tasks is 16% (not shown in the table). We find that the impact of refactorings on each project can be substantial with max S^{ref} of 97%. Note that our experiments perform *one* refactoring at a time, and the impact of multiple refactorings is likely to be higher because they are likely to modify more classes.

5.2.7 ***RQ3****: How many tests does Reks skip on average for various refactoring types if refactorings are systematically performed?* We measured the average S^{ref} per refactoring type (across all projects) for all refactoring types available in Eclipse IDE. Table 3 shows the numbers for each refactoring type. (Max values are similar because many refactoring types impact many tests in `Pebble`.) Introduce Param. Obj., on average, has the biggest impact on regression tests (with S^{ref} of 27%, followed by Change Signature (24%) and Introduce Indirection (24%). On the other side, we find that Infer Generic Type Args has the smallest impact on regression testing (with S^{ref} of 0.05%), followed by Rename Local (0.70%);

Table 3: Percent of Test Classes Not Run Due to Reks Per Refactoring Type (Systematically Performed Refactorings).

	Refactoring	S^{ref} Max	Med	Avg	STD
1	Rename Field	96.67	3.73	13.04	20.42
2	Rename Method	96.67	10.00	22.32	27.12
3	Rename Local	96.67	0.00	0.70	2.91
4	Move Method	96.67	7.46	13.69	13.90
5	Change Signature	96.67	10.00	23.65	29.11
6	Extract Method	96.67	10.00	19.27	24.07
7	Extract Local	96.67	9.43	18.01	22.51
8	Extract Constant	96.67	7.98	15.00	21.22
9	Inline Constant	96.67	3.00	9.16	15.96
10	Inline Method	96.67	2.52	11.95	20.87
11	Inline Local	96.67	4.48	18.48	28.08
12	Convert Local to Field	96.67	4.00	18.11	27.95
13	Convert Anonymous	96.67	7.46	22.21	28.78
14	Move Type to New File	72.13	2.99	9.17	14.71
15	Extract Superclass	96.67	8.82	16.85	24.35
16	Extract Interface	96.67	2.38	9.59	18.55
17	Use Supertype	96.67	0.00	2.50	11.86
18	Push Down	96.67	0.00	2.15	10.71
19	Pull Up	96.67	14.29	21.10	23.55
20	Extract Class	96.67	5.00	18.97	26.34
21	Introduce Param. Obj.	96.67	10.34	27.01	32.42
22	Introduce Indirection	96.67	9.52	23.61	27.68
23	Introduce Factory	96.67	2.52	15.01	25.90
24	Introduce Parameter	96.67	2.38	9.29	18.33
25	Encapsulate Field	96.67	2.38	12.04	21.60
26	Generalize Type	96.67	3.00	15.75	25.24
27	Infer Generic Type Args	12.58	0.00	0.05	0.60
	Max	96.67	14.29	27.01	32.42
	Avg	92.64	N/A	14.40	N/A

Table 4: Execution Time for Reks, RetestAll, and Ekstazi.

Project	Reks [s]	All [s]	Ekstazi [s]	R/A [%]
Coll	22.95	59.01	34.57	38.90
Config	14.32	54.14	33.80	26.46
DBCP	6.88	86.71	35.77	7.93
IO	4.34	132.69	11.92	3.27
JClassmate	2.85	3.45	4.15	82.80
JObjectDiff	19.93	35.62	35.67	55.95
Lang	14.85	43.45	17.85	34.17
Net	3.87	63.07	4.93	6.13
Pebble	1.20	6.20	5.90	19.42
Stateless4J	1.77	2.24	2.75	79.03
Max	22.95	132.69	35.77	82.80
Avg	9.30	48.66	18.73	35.41

renaming a local variable rarely (only if used in an anonymous class) modifies classfiles, which are collected by Reks as dependencies.

5.2.8 ***RQ4****: What is the cost of the Reks update rules and how does this cost compare to the test execution time?* We measured (Table 4) the update time for refactoring types that recompute cksum of dependencies. To measure the time, we altered the procedure in Figure 12 to invoke the update rules during Maven build on line 16. Specifically, we compare the time to build project with Maven that invokes update rules (second column) with the time to run all tests during the Maven build, i.e., RetestAll (third column), and time to build project with Maven with Ekstazi (fourth column). Update rules take ~1sec for all projects and the rest is taken by Maven. The table also shows the ratio of Reks and RetestAll (the last column).

5.2.9 ***RQ5****: How many test methods fail, on average, due to refactoring tasks (performed by the Eclipse refactoring engine)?* We proposed (see Section 1) that Reks should be a part of the pre-submit testing phase, because we are aware that the existing refactoring engines do not always preserve program behavior [27, 38, 39, 66]. We measured the average percent of failing test methods as the number of failing test methods / total number of test methods. To obtain this measure we modified line 16 in Figure 12 to *run* the tests (rather than to invoke the analysis). The average percent of failing test methods was only 0.47%; we first computed the value for each project and then the average across all projects. In the future, we believe that formally proven refactoring transformations [7, 14, 62] will enable the use of Reks in the post-submit phase.

6 DISCUSSION

RTS granularity. Reks rules are applicable only to RTS techniques that collect dependencies on classes [28, 41, 42, 55, 64]. The rules for RTS techniques with different selection granularity would differ. Consider TestTube [13], an RTS techniques that collect dependencies on methods [55]. The modification rules for TestTube can be more precise than the rules for Ekstazi, e.g., if a method being moved from one class ($\hat{c}$) to another ($\hat{c}'$) is the only method used by a test, then there is no reason for the test to depend on $\hat{c}$ after the refactoring task. At the same time, rules for some refactoring types would be more complex. Consider an invocation of Extract Method on code inside a method `m` to a new method `n`. The rules for TestTube would have to add method `n` in the set of dependencies for tests that already depend on `m`, because TestTube does not know if the extracted code was executed by a test. This overapproximation is similar to Move Method refactoring in Reks. We plan to explore the update rules for other RTS techniques in the future.

Open-source vs. systematic refactorings. The difference in average savings is due to the difference in the set of refactoring types. Some refactoring types that are performed only systematically (e.g., Infer Generic Type Args) have low impact on tests, and some refactoring types (e.g., rename class), which have big impact on tests, are only performed in open-source projects.

Safety. The Reks technique is based on two assumptions: refactoring transformations are behavior-preserving and Ekstazi is safe. We discussed the former in detail in Section 1. As for the latter, a few researchers semi-formally proved safety of RTS techniques [64]. Future work should formally prove correctness of RTS techniques and develop a testing framework to check safety of RTS tools.

7 THREATS TO VALIDITY

External. Our results might not generalize beyond the projects used in the study. To mitigate this threat, we used projects of different sizes, which are also used in different application domains. We also evaluated the impact of refactorings on regression testing using a large number of refactorings in open-source projects.

We have evaluated our rules on the refactoring types that are available in the Eclipse JDT. However, many popular refactoring types (e.g., rename) are shared among all IDEs, so our rules should be directly applicable. In the future, we plan to study if Reks is applicable on refactorings that are unique to other IDEs.

Internal. Reks and our scripts may contain bugs, which may impact our conclusions. We used meta information collected during our procedure (Figure 12) to manually check the results for a large number of refactorings. With unit tests, we confirmed that Reks and Ekstazi behave the same for non-refactoring changes.

Construct. When systematically applying refactorings, we measured the impact of one refactoring at a time. Our goal was to evaluate the impact of each refactoring separately, but evaluating the impact of the combination of refactorings is an interesting future research direction. We also limit the number of code elements that we use in each compilation unit to make the experiment feasible.

8 RELATED WORK

There has been a lot of work on regression test selection [6, 21–23, 28, 44, 49, 54, 55, 59, 60, 68, 70, 77–80] and refactorings [24, 51, 52, 73]. However, prior work mostly explored these two topics independently. We discuss work on (1) combining regression testing and refactorings [57], (2) testing refactorings [16, 27, 47, 66], (3) RTS techniques, and (4) refactorings.

Regression testing guided by refactorings. Rachatasumrit and Kim [57] found that existing regression tests are inadequate to validate the correctness of refactorings. They evaluated the quality of regression tests for three real-world Java applications. The results showed that only 22% of changes are covered by the tests. Additionally, the results showed that only 38% of regression tests cover at least one refactoring change. Mongiovi et al. [46] proposed an approach to improve regression test suites via automated test generation. Their technique is guided by the coverage of refactored changes. They developed two tools—Safira, which identifies the changes in program elements, and SafeRefactorImpact, which generates random tests using Randoop [56]—with the goal to exercise the changed elements. In a follow-up work, Soares et al. [67] implemented a prototype tool that runs concurrently with an IDE and generates tests to cover the changed code, while the code is being edited. Recently, Alves et al. [1], presented a test suite prioritization technique to faster detect bugs introduced by refactorings.

Unlike prior work, we systematically evaluated the impact of refactorings on RTS and proposed update rules to modify the dependencies for each test class, with the goal to avoid unnecessary test executions for behavior-preserving transformations [14, 62].

Testing manual and automated refactorings. Alves et al. [2] implemented a code review tool called RefDistiller (which consists of RefChecker and RefSeparator) that detects anomalies introduced by manual refactoring edits. GhostFactor [26] is similar to RefDistiller in that it aims to check the correctness of manual refactorings automatically using a set of predefined conditions. Our work targets automated transformations and speeds up regression testing.

A lot of work was done on automated testing of refactoring engines. Daniel et al. [16] proposed a syntax tree generator, called ASTGen, which systematically generates a large number of Java programs based on the given imperative description. Gligoric et al. [27] introduced RTR, which systematically applies refactorings on the given set of applications. The results showed that the failure rate for the Eclipse refactoring engine (due to compilation errors) was 1.4% for JDT. Soares et al. [66] followed a similar approach with ASTGen to generate a set of test inputs exhaustively to detect bugs that modify the program behavior.

Reks differs from the prior work as the main purpose of Reks is to speed up regression testing by skipping tests that are affected only by refactorings. In cases when an automated refactoring introduce a compilation error or a refactoring task throws an exception, the developer would not run the tests and Reks does not update dependencies. Because of changes that modify program behavior, we suggest to integrate Reks in the pre-submit testing phase [20].

Regression test selection. There has been several decades of research on RTS. Several surveys nicely describe various aspects of RTS [6, 22, 79]. Most of the RTS techniques differ in the granularity on which they collect dependencies. Google TAP [10] keeps dependencies among projects. Ekstazi [28] collects dependencies on files. FaultTracer [80] and TestTube [13] collect dependencies on methods, Echelon [68] collects dependencies on basic blocks, etc. To the best of our knowledge, Reks is the first refactoring-aware RTS technique.

Refactorings. Similarly to RTS, there was a lot of work related to refactorings. Researchers have explored refactorings for various languages and domains [4, 5, 8, 9, 15, 32, 43, 45, 69, 76]. Closely related is work by Kim et al. [40] that introduced RefFinder, a tool to detect manual refactorings in code repositories. We evaluated Reks on a large number of refactorings performed by open-source developers. In the future, we can deploy RefFinder to help us detect revisions of interest.

9 CONCLUSIONS

We presented the first step toward refactoring-aware RTS technique, called Reks. Unlike the existing RTS techniques that run tests regardless of a change, Reks specially treats code changes made by automated behavior-preserving transformations (i.e., refactoring). When a refactoring is performed, Reks updates the dependencies for the affected tests and does not run any test. Based on our study of refactorings performed by developers of open-source projects, Reks does not run, on average, 33% of available tests (which would be run with a refactoring-unaware technique). Also, based on our systematic study on several popular open-source projects, we find that Reks can avoid running up to 97% tests (16% on average). We integrated Reks with Eclipse via a plugin, which helps developers to save regression testing time in the pre-submit testing phase. Although we currently expect that developers run all tests in the post-submit testing phase, our work present a way in which formally proven refactorings and RTS could work in the future in all testing phases.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments; Farah Hariri, Sarfraz Khurshid, Owolabi Legunsen, Shirley Liu, Darko Marinov, Aleksandar Milicevic, and August Shi for their feedback on this work. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363 and CCF-1652517, and by a Google Faculty Research Award.

REFERENCES

[1] Everton L. G. Alves, Patrícia D. L. Machado, Tiago Massoni, and Miryung Kim. 2016. Prioritizing Test Cases for Early Detection of Refactoring Faults. *Software Testing, Verification and Reliability* (2016), 402–426.

[2] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits. In *International Symposium on Foundations of Software Engineering*. 751–754.

[3] Apache Camel - Building 2017. Building Apache Camel. (2017). http://camel.apache.org/building.html.

[4] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking Rewriting on Java. In *Rewriting Techniques and Applications*. 36–47.

[5] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. 2004. DMS: Program Transformations for Practical Scalable Software Evolution. In *International Conference on Software Engineering*. 625–634.

[6] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica (Slovenia)* 35, 3 (2011), 289–321.

[7] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. 2004. Algebraic Reasoning for Object-oriented Programming. *Sci. Comput. Program.* 52, 1-3 (2004), 53–100.

[8] Marat Boshernitsan and Susan L. Graham. 2004. iXj: Interactive Source-to-Source Transformations for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 212–213.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72, 1-2 (2008), 52–70.

[10] Build in the Cloud 2017. Build in the Cloud: How the Build System works. (2017). http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.

[11] Byte Buddy 2017. Byte Buddy. (2017). https://github.com/raphw/byte-buddy.

[12] Lianping Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *Software* 32, 2 (2015), 50–54.

[13] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A System for Selective Regression Testing. In *International Conference on Software Engineering*. 211–220.

[14] Julien Cohen. 2016. Renaming Global Variables in C Mechanically Proved Correct. In *International Workshop on Verification and Program Transformation*. 50–64.

[15] James R. Cordy. 2006. The TXL Source Transformation Language. *Science of Computer Programming* 61, 3 (2006), 190–210.

[16] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Fundamental Approaches to Software Engineering*. 185–194.

[17] EclipseJDTWebPage 2017. Eclipse Java development tools (JDT). http://eclipse.org/jdt. (2017).

[18] EclipseWebPage 2015. Eclipse Indigo. (2015). https://eclipse.org.

[19] Ekstazi 2017. Ekstazi: Lightweight Test Selection. (2017). http://www.ekstazi.org.

[20] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.

[21] Emelie Engström and Per Runeson. 2010. A Qualitative Survey of Regression Testing Practices. In *Product-Focused Software Process Improvement*. 3–16.

[22] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Journal of Information and Software Technology* 52, 1 (2010), 14–30.

[23] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In *International Symposium on Empirical Software Engineering and Measurement*. 22–31.

[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Adison-Wesley.

[25] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should We Control?. In *International Conference on Software Engineering*, Vol. 1. 55–65.

[26] Xi Ge and Emerson Murphy-Hill. 2014. Manual Refactoring Changes with Automated Refactoring Validation. In *International Conference on Software Engineering*. 1095–1105.

[27] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. 2013. Systematic Testing of Refactoring Engines on Real Software Projects. In *European Conference on Object-Oriented Programming*. 629–653.

[28] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.

[29] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *Automated Software Engineering*. 361–372.

[30] Jean Hartmann. 2007. Applying Selective Revalidation Techniques at Microsoft. In *Pacific Northwest Software Quality Conference*. 255–265.

[31] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *International Conference on Software Engineering*. 483–493.

[32] Mark Hills, Paul Klint, and Jurgen J. Vinju. 2012. Scripting a Refactoring with Rascal and Eclipse. In *Workshop on Refactoring Tools*. 40–49.

[33] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Automated Software Engineering*. 426–437.

[34] IntelliJIdeaWebPage 2017. IntelliJ IDEA. http://www.jetbrains.com/idea. (2017).

[35] JaCoCoWebPage 2017. JaCoCo Java Code Coverage Library. (2017). http://eclemma.org/jacoco/.

[36] JDTRefactoringMenuWebPage 2017. Eclipse JDT Refactorings Menu. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm. (2017).

[37] JDTRefactoringWebPage 2017. Eclipse JDT Refactoring Support. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm. (2017).

[38] Jongwook Kim, Don Batory, and Danny Dig. 2015. Scripting Parametric Refactorings in Java to Retrofit Design Patterns. In *International Conference on Software Maintenance and Evolution*. 211–220.

[39] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. 2016. Improving Refactoring Speed by 10X. In *International Conference on Software Engineering*. 1145–1156.

[40] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-finder: a Refactoring Reconstruction Tool Based on Logic Query Templates. In *International Symposium on Foundations of Software Engineering*. 371–372.

[41] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. *Journal of Object-Oriented Programming* 8, 2 (1995), 51–65.

[42] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594.

[43] Huiqing Li and Simon Thompson. 2012. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering*. 501–515.

[44] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-World Software Evolution?. In *International Conference on Software Engineering*. 535–546.

[45] Tom Mens and Tom Tourwe. 2001. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *International Conference on Software Maintenance*. 570–579.

[46] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making Refactoring Safer Through Impact Analysis. *Science of Computer Programming* 93 (2014), 39–64.

[47] Michael Mortensen, Sudipto Ghosh, and James M. Bieman. 2006. Testing During Refactoring: Adding Aspects to Legacy Systems. In *International Symposium on Software Reliability Engineering*. 221–230.

[48] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *International Conference on Software Engineering*. 287–297.

[49] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based Regression Test Case Selection, Minimization and Prioritization: A Case Study on an Industrial System. *Journal of Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.

[50] NetBeansWebPage 2017. NetBeans. https://netbeans.org. (2017).

[51] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[52] William F. Opdyke and Ralph E. Johnson. 1990. Refactoring: An Aid In Designing Application Frameworks and Evolving Object-Oriented Systems. In *Symposium on Object-Oriented Programming Emphasizing Practical Applications*. 145–161.

[53] OracleJDeveloperWebPage 2017. Oracle JDeveloper. http://www.oracle.com/technetwork/developer-tools/jdev. (2017).

[54] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Future of Software Engineering*. 117–132.

[55] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *International Symposium on Foundations of Software Engineering*. 241–251.

[56] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering*. 75–84.

[57] Napol Rachatasumrit and Miryung Kim. 2012. An Empirical Investigation into the Impact of Refactoring on Regression Testing. In *International Conference on Software Maintenance*. 357–366.

[58] RefactoringCatalogWebPage 2017. Catalog of Refactorings. http://refactoring.com/catalog/. (2017).

[59] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 432–448.
[60] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *Transactions on Software Engineering* 22, 8 (1996), 529–551.
[61] David Saff and Michael D. Ernst. 2004. An Experimental Evaluation of Continuous Testing During Development. In *International Symposium on Software Testing and Analysis*. 76–85.
[62] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Challenge Proposal: Verification of Refactorings. In *Workshop on Programming Languages Meets Program Verification*. 67–72.
[63] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *International Symposium on Foundations of Software Engineering*. 858–870.
[64] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *International Journal of Software Engineering and Knowledge Engineering* 17, 3 (2007), 359–378.
[65] SloccountWebPage 2017. SLOCCount. (2017). http://www.dwheeler.com/sloccount.
[66] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *Transactions on Software Engineering* 39 (2013), 147–162.
[67] Gustavo Soares, Emerson Murphy-Hill, and Rohit Gheyi. 2013. Live Feedback on Behavioral Changes. In *International Workshop on Live Programming*. 23–26.
[68] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *International Symposium on Software Testing and Analysis*. 97–106.
[69] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. 2011. A Refactoring Constraint Language and its Application to Eiffel. In *European Conference on Object-oriented Programming*. 255–280.
[70] TestingAtSpeedAndScaleOfGoogleWeb 2011. Testing at the Speed and Scale of Google. (2011). http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[71] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3 (2011), 9:1–9:47.
[72] Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for Generalization using Type Constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 13–26.
[73] Lance Tokuda and Don Batory. 1999. Evolving Object-Oriented Designs with Refactorings. In *Automated Software Engineering*. 174–181.
[74] ToolsForContinuousIntegrationAtGoogleScaleWeb 2011. Tools for Continuous Integration at Google Scale. (2011). http://www.youtube.com/watch?v=b52aXZ2yi08.
[75] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and Back Again: Can you Compile that Snapshot? *Journal of Software: Evolution and Process* (2017).
[76] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. 2001. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. 365–370.
[77] G. Wikstrand, R. Feldt, J. K. Gorantla, W. Zhe, and C. White. 2009. Dynamic Regression Test Selection Based on a File Cache - An Industrial Evaluation. In *International Conference on Software Testing, Verification, and Validation*. 299–302.
[78] Guoqing Xu and Atanas Rountev. 2007. Regression Test Selection for AspectJ Software. In *International Conference on Software Engineering*. 65–74.
[79] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.
[80] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing Failure-Inducing Program Edits Based on Spectrum Information. In *International Conference on Software Maintenance*. 23–32.