2016 IEEE 27th International Symposium on Software Reliability Engineering

The Effect of Test Suite Type on Regression Test Selection

Nima Dini[1], Allison Sullivan[1], Milos Gligoric[1], and Gregg Rothermel[2]
The University of Texas at Austin[1], University of Nebraska - Lincoln[2]
Email: {nima.dini, allisonksullivan, gligoric}@utexas.edu, grother@cse.unl.edu

***Abstract*—Regression test selection (RTS) techniques reduce the cost of regression testing by running only test cases related to code modifications. RTS techniques have been extensively researched, and the effects of several context factors on techniques have been empirically studied, but no prior work has explored the effects that might arise due to differences in types of test suites. We believe such differences may matter, and thus, we designed an empirical study to investigate them. Specifically, we consider two types of test suites obtained with automated test case generation techniques—feedback-directed random techniques and search-based techniques—along with manually written test suites. We assess the effects of these test suite types on two RTS techniques: a "fine-grained" technique that selects test cases based on dependencies tracked at the method level and a "coarse-grained" technique that selects test cases based on dependencies tracked at the file level. We performed our study on eight open-source projects across 800 commits. Our results show that on average, fine-grained RTS was more effective for test suites created by search-based test case generation techniques whereas coarse-grained RTS was more effective for test suites created by feedback-directed random techniques, and that commits affect RTS techniques differently for different types of test suites.**

I. Introduction

Regression testing is performed with the goal of assessing whether code modifications have harmed previously working functionality. Although important, regression testing is costly, and the cost is reportedly increasing. For example, Microsoft reports that the annual cost of regression testing exceeds tens of millions of dollars [35], [36], and Google reports a quadratic increase in regression testing time due to a linear increase in the number of commits per day and a linear increase in the number of test cases added per commit [21], [63], [65]. This cost will arguably increase further as companies adopt *automated test case generation techniques* (e.g., [6], [17], [27], [31], [49], [58], [59]) to augment manually written test suites.

Regression test selection (RTS) techniques (e.g., [3], [5], [14], [22]–[24], [44], [46], [47], [53], [55], [61]–[63], [69], [72], [74], [75]) reduce the cost of regression testing by omitting test cases that are thought to be unnecessary to test a given modified version of a program. Coverage-based RTS techniques track *test case dependencies* (i.e., statements, methods, or files covered by test cases) and select only test cases for which dependencies are potentially affected by code modifications, reducing test execution time.

RTS techniques have been extensively researched and empirically studied. Several empirical studies have considered the effects of various context factors on the relative cost-effectiveness of techniques. These include the effects of test case granularity (e.g., a measure of test suite size and complexity) [3], the effects of RTS technique granularity (the level of coverage at which techniques operate) [9], and the effects of types of modifications on test selection [20]. These studies have shown that context factors *can* affect the relative cost-effectiveness of techniques, with implications for which techniques practitioners should apply in certain situations. To date, however, no work has attempted to examine *the effects that types of test suites may have on RTS techniques*.

We believe that the use of different types of test suites *can* affect the relative performance of RTS techniques. Further, with the increased adoption of automated test case generation techniques, such differences are likely to become increasingly important. We thus designed an empirical study to evaluate the effect of types of test suites on the effectiveness of RTS techniques. We consider two types of test suites obtained by automated test case generation techniques: *feedback-directed random test case generation techniques* [49], and *search-based test case generation techniques* [25], [64]. For brevity, we refer to test suites created by the former as RGTs, and test suites created by the latter as SGTs (we sometimes refer to RGTs and SGTs collectively as GTs). We also use *manually written test suites* (MWTs). We evaluate the impact of these three types of test suites on two RTS techniques: a "fine-grained" RTS technique that tracks test case dependencies on methods [9] and a "coarse-grained" RTS technique that tracks test case dependencies on files [30]. To measure technique effectiveness we compute the *test selection ratio* (i.e., the number of selected test cases divided by the total number of test cases) for each combination of RTS and type of test suite.

Our study addresses three research questions:

RQ1: How effective are RTS techniques when applied to various types of test suites?

RQ2: Does the effectiveness of RTS techniques vary across types of test suites?

RQ3: Do differences in RTS technique performance across types of test suites vary across commits?

RQ1 is not itself the primary focus of this work, but we do need to investigate it, because if RTS techniques are not effective in the context of our study setting, then there is no point in comparing them further. RQ2 is the primary focus of this work. RQ3 becomes important if differences are observed in relation to RQ2, because in that case it could be useful to determine the extent to which differences observed in relation to RQ2 vary across individual commits.

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.28

To address our research questions we study the application of our two RTS techniques across the three different types of test suites for eight open-source projects, and we study these across sequences of 100 commits for each of these projects. Our empirical results show that in the cases considered, RTS techniques achieved (on average) test selection ratios between 13% and 32%. Further, on average, for all types of test suites considered, the fine-grained RTS technique was about twice as effective as the coarse-grained technique in terms of test selection ratio. Comparing results across the types of test suites, we find that types of test suites do affect RTS technique effectiveness, and they affect the two types of RTS techniques that we study differently. For example, on average, the fine-grained RTS technique was more effective than the coarse-grained technique for SGTs, whereas the coarse-grained RTS technique was more effective than the fine-grained technique for MWTs and RGTs. Finally, we find that commits also affect the performance of RTS techniques, and do so differently for different types of test suites. For example, there were a large number of commits for which only MWTs were selected, and several commits for which only GTs were selected.

There are several implications of our results. First, if our results generalize, they suggest that developers can be confident that the cost of regression testing will not substantially increase as developers add automatically generated test cases. Second, our study shows that a developer may not want to choose just one RTS technique for their entire code base, but rather, may do better by using different RTS techniques for different types of test suites. Third, our results may be useful for developers who wish to make decisions about which types of test cases to include in a regression test suite. Finally, our results have implications for test case design and the grouping of test cases into clusters, suggesting approaches for both that can promote more effective regression test selection.

II. Study Setup and Methodology

This section presents our objects of study, variables, experiment procedure, and threats to validity.[1]

A. Objects of Study

As objects of study we used eight projects. These projects had all been harvested for and utilized in earlier studies of regression testing [30], [34], [60]. Originally, these objects of study were chosen with the following requirements in mind: (1) the project has at least 100 existing MWTs (JUnit test methods), (2) the project's most recent commit (at the time it was harvested) builds without errors, (3) the project has at least 100 commits, (4) the project uses Maven to build code and run test cases, and (5) the project is available on GitHub. The first two requirements were necessary to ensure that projects were compilable and non-trivial and allowed us to work with test cases created by actual developers, the third allowed us to observe results across a sufficiently large project history, and the last two were necessary to enable automation of the empirical studies. In addition, we required that RGTs and SGTs generated for what we considered an initial commit (i.e., the commit that is 100 commits prior to the most recent commit) compile without errors.

[1]The tools used and data collected in this study can all be obtained by contacting the first author.

TABLE I: Objects of Study

Project	SHA	LOC	MWTs		Time
			method	class	[ms]
Codec	535bd812	17625	693	48	24594
Coll	c87eeaa4	60251	15435	159	51390
DBCP	e86d1c2e	19908	541	29	103363
Lang	17a6d163	69014	3685	134	47640
Math	471e6b07	174832	5825	431	160631
Net	4450add7	26928	265	42	79160
Pool	14eb6188	13352	270	20	302224
Vectorz	2d6d264f	50905	424	70	14606
$\sum$	n/a	432815	27138	933	783608

Table I provides basic data for each of our objects of study, including a brief name, the identifier of the most recent commit (i.e., Git SHA) available at the time we harvested the project, the number of lines of code in the most recent commit, the numbers of test methods and test classes available in the most recent commit, and the time (in milliseconds) required to execute the MWTs that came with the projects. `Codec` [10] provides implementations of common encoders and decoders; `Coll` [11] extends and augments the Java Collections Framework; `DBCP` [15] supports interactions with a relational database; `Lang` [39] extends the standard Java library; `Math` [41] includes mathematics and statistics components; `Net` [45] implements many internet protocols; `Pool` [50] provides an object-pooling API and a number of object pool implementations; and `Vectorz` [66] is a fast double-precision vector and matrix library for Java.

B. Variables

1) Independent Variables: We manipulate two independent variables: test suite type and RTS technique. As noted in Section I, the test suites we utilize include test suites generated by two automated test case generation techniques (a feedback-directed random technique and a search-based technique), and the manually written test suites that were provided with our objects of study. Our RTS techniques include one "fine-grained" technique that tracks dependencies at the level of methods and one "coarse-grained" technique that tracks dependencies at the level of files. As particular instantiations of techniques we relied on existing tools, as follows:

Feedback-Directed Random Test Case Generation: *Randoop* [48], [49], [52], [54] implements a feedback-directed random test case generation technique. Unlike traditional random techniques, which naïvely create sequences of method calls, Randoop uses the outcome of each sequence to guide the test case generation process. Specifically, Randoop begins the test case generation process with a set of default values of each type and an empty set of sequences of method calls. In each step, Randoop randomly chooses a next method to invoke (from one of the classes under test) and randomly chooses arguments to pass to the method; these arguments

are derived either from the set of default values or the results of some prior sequence of method calls. Randoop executes the new sequence and makes decisions based on the outcome. First, sequences that throw exceptions are not further extended, because the result of the extended sequence would be the same regardless of the extension. Second, sequences that violate user-defined contracts are not converted to test cases and not used in further steps. Finally, sequences that are subsumed by the new sequence are discarded. Randoop outputs the final set of sequences of method calls as JUnit4 test cases.

Search-Based Test Case Generation: *EvoSuite* [25]–[27], [57] implements a search-based test case generation technique [64]. EvoSuite randomly generates an initial population of test cases and then iteratively generates test cases by evolving the initial population using mutation, selection, and crossover operators. A user can customize aspects of the tool including execution time (the default is two minutes per class under test) and the fitness function used to guide the search; the default fitness function attempts to maximize branch coverage. The tool outputs the test suite with the highest coverage. Similar to Randoop, EvoSuite outputs test cases in JUnit4 format. Unlike Randoop, EvoSuite minimizes the generated test suites with respect to a coverage criterion.

Coarse-Grained Regression Test Selection: *Ekstazi* [19], [28]–[30] is an RTS tool for Java that tracks *dynamic* dependencies of test classes[2] on *files*; the files can be either executable code or external resources, such as a property file. A test class need not be run for a new commit if none of its dependent files has changed. Unlike prior RTS techniques that track dependencies at finer-grained levels (e.g., statements, basic blocks, or methods), Ekstazi tracks dependencies at a more coarse-grained level. Prior work on Ekstazi showed that it can reduce regression testing cost by more than 50% on non-trivial projects using MWTs [28], [30]. Ekstazi further reduces the number of selected test cases at each commit by using smart checksums, which find two Java class files to be the same if the code modifications they contain are only in parts of the files that are not exercised by test cases. Ekstazi has been used by several open-source projects, including Apache Camel, Apache CXF, and Apache CommonsMath.

Fine-Grained Regression Test Selection: *TestTube* [9] is an RTS tool that tracks dependencies at the level of *functions* (which in our context maps to *methods*) rather than files. TestTube was implemented for the C programming language; in this work we implemented the algorithm within the same framework for Java used to implement Ekstazi. The key difference between the original implementation and ours is that ours groups test cases with the same name; this approach was taken to make the tool practical. In other words, if a test case `t` executes a method `m` in class `C`, TestTube records that `t` depends on all methods named `m` in class `C` (regardless of their signature). If any method `m` changes, TestTube selects `t` for the execution. Note that our implementation of TestTube may be somewhat less precise than the original TestTube technique.

[2]We use "test case" and "test class" interchangeably; to refer to individual test cases when needed, we use "test method".

```
Input: project under study
 1: function CollectData(project)
 2:     Clone(project.url)
 3:     Checkout(project.latest)
 4:     SetJava(project.jversion)
 5:     Checkout(project.initial)
 6:     GTs ← EvoSuite/Randoop(project)
 7:     for all commit ∈ Last100Commits(project)
 8:         CleanAndCheckout(project, commit)
 9:         if RunBuild(project) == FAIL
10:             continue
11:         end if
12:         project.tests ← GTs
13:         testCompileError ← RunBuild(project)
14:         GTs ← GTs \ testCompileError
15:     end for
16:     project.tests ← GTs
17:     for all commit ∈ Last100Commits(project) do
18:         CleanAndCheckout(project, commit)
19:         if RunBuild(project) == FAIL then
20:             continue
21:         end if
22:         ReplaceExitWithException(project)
23:         RunTest(project)                    ▷ Run available tests
24:         IntegrateEkstazi(project)
25:         RunTest(project)                    ▷ Run selected tests
26:         IntegrateTestTube(project)
27:         RunTest(project)                    ▷ Run selected tests
28:     end for
29: end function
```

Fig. 1: CollectData procedure that obtains the data to be analyzed in the study. Highlighted statements are executed only for RGTs and SGTs

2) Dependent Variable: Our dependent variable measures the effectiveness of RTS techniques at reducing the amount of effort required to retest modified programs. To do this we compute the *test selection ratio* for the technique when applied to a given object program, version, and test suite: this is calculated as the number of test cases selected divided by the total number of test cases in the test suite. We denote this variable by $\mathcal{S}^X$, where X denotes the type of test cases the ratio corresponds to (MWT, RGT, or SGT).

C. Experiment Procedure

For each of our objects of study we wished to evaluate the effectiveness of Ekstazi and TestTube for MWTs, RGTs, and SGTs. Figure 1 provides the procedure we used to collect the data for the analysis on a single object of study; we performed this procedure for each object of study. In this section we describe the approaches used to collect data for various test suites. We then describe the approaches we used to configure Ekstazi, TestTube, Randoop, and EvoSuite and provide data on the test cases generated by Randoop and EvoSuite. Finally, we describe the platform used for the study.

1) Collecting Data for Test Suites: We first describe the steps taken to collect data for MWTs (Figure 1 without the highlighted statements) and then describe modifications to

those steps needed to collect the data for RGTs and SGTs (Figure 1 with the highlighted statements).

MWTs: In the first step of our procedure (Lines 2 and 3 in Figure 1), we clone the project from GitHub and check out its most recent commit. Next, we obtain the appropriate Java version for the project (Line 4); of the eight projects only `Coll` and `Lang` require Java 7, while others work with Java 8. Next (Lines 17-28), moving from older towards newer commits, we iterate over the latest 100 commits. We use only commits on the master branch (obtained by `git log --first-parent --no-merges`), because most of the projects run regression tests only on those commits. For each commit, we revert the existing changes (introduced by prior iterations of our procedure) and then attempt to build the project. If the build fails, we move on to the next commit, otherwise we execute the following actions: (a) run *all* available test cases, (b) run test cases *selected* by Ekstazi, and (c) run tests cases *selected* by TestTube. We collect the number of test cases executed for (a), (b), and (c). Using the number of available and selected test cases, we calculate the test selection ratios, $\mathcal{S}^{MWT}$, for the techniques. Additionally, for each commit, we save dependency information tracked by Ekstazi and TestTube; this information is used by Ekstazi and TestTube in subsequent commits and our subsequent data analysis.

RGTs and SGTs: Our steps for collecting data for GTs diverge somewhat from the steps described above and include all statements highlighted in Figure 1. The first steps (cloning and setting up Java) remain the same. In the second step (Lines 5-16), we use Randoop or EvoSuite to generate test cases for the initial commit. Then, we compile the generated test cases across all 100 commits and remove any test case that does not compile in at least one commit. In other words, we require test cases to compile for all commits used in our study. We do this because test case generation can be expensive [32], [43], [67] and therefore it is unlikely that newly generated test cases will be added at every commit. In the third step we begin from the initial commit and then move forward one commit at a time to execute the following actions: (a) run all GTs, (b) run GTs selected by Ekstazi, and (c) run GTs selected by TestTube. We collect the same data as we do for MWTs and compute the test selection ratios $\mathcal{S}^{RGT}$ and $\mathcal{S}^{SGT}$. We also replace "`System.exit`" in the code under test with "`if (true) throw new Error();`" prior to executing any GT to avoid interrupting the Java Virtual Machine.

2) Configuring Systems: ***Ekstazi and TestTube:*** We use Ekstazi version 4.6.1, available on Maven Central, with its default configuration options. As noted earlier, we implemented TestTube by extending the same version of Ekstazi.

Randoop and EvoSuite: We use a recent version of Randoop (SHA `ce2b25c6`) available on GitHub. For the initial questions (introduced in Section I), we used the default configuration of Randoop. In Section IV, we discuss the impact of several configuration options on our results. Note that Randoop, in the default configuration, generates test cases for 100 seconds for all classes under test.

TABLE II: Numbers of Compiled/Generated GTs

Project	RGT	SGT	RGT^b	RGT^c	RGT^d
Codec	2526 / 2527	1022 / 1094	2422 / 2423	9026 / 9028	7094 / 7095
Coll	733 / 794	-	1364 / 1553	3708 / 4371	3709 / 4233
DBCP	14276 / 14277	154 / 1513	13517 / 13518	21054 / 21055	14583 / 14584
Lang	2881 / 2881	-	3371 / 3371	6858 / 6859	4544 / 4545
Math	1284 / 1640	2644 / 4588	1268 / 1689	1004 / 1196	1951 / 2192
Net	1511 / 1511	2333 / 2635	5367 / 5367	2151 / 2151	1316 / 1316
Pool	8326 / 8326	164 / 460	9205 / 9205	16600 / 16600	11655 / 11655
Vectorz	1871 / 2135	8257 / 9331	2527 / 3465	2961 / 3192	2077 / 2269
$\sum$	33408 / 34091	14574 / 19621	39041 / 40591	63362 / 64452	46929 / 47889

We use EvoSuite version 1.0.2 in its default configuration. By default, EvoSuite generates test cases for two minutes for each class under test.

As inputs to Randoop and EvoSuite we provide a list of all (non-test) classes available in each project. For each Randoop configuration we generated one test method per test class (`--testsperfile=1`); the default is 500 test methods per test class. We also generated one test method per test class in EvoSuite by post-processing the generated test cases. Using one test method per test class was necessary to ensure that we keep all test methods that compile across 100 commits; otherwise we would discard a substantial number of test methods. We discuss the impact of this decision in Section V.

3) Data on RGTs and SGTs: Table II shows the numbers of test cases generated by Randoop and EvoSuite in Columns 2 and 3, respectively (we postpone discussion of Columns 4–6 to Section IV). The numbers below the lines are the numbers of test cases generated for the initial commits and the numbers above the lines are the numbers of test cases that successfully compiled across all 100 commits. A substantial number of test cases could be compiled, so we did not repair broken test cases [13], [42]. We did not generate test cases with EvoSuite for `Coll` and `Lang` because these projects do *not* build with Java 8 and used version of EvoSuite required Java 8.

Randoop generated unusually large numbers of test cases for `DBCP` and `Pool`. We found that these projects contain numerous getter and setter methods. This resulted in the generation of many short sequences of method calls, because invoking a getter requires only a target object on which the method is invoked and invoking a setter requires only a target object and an argument (which can simply be the default value).

Table III shows the branch and class coverage obtained on the objects of study using MWTs, RGTs, and SGTs as reported by JaCoCo [37]. In most cases, MWTs achieved higher branch coverage than RGTs and SGTs (`Net` is an exception as it

TABLE III: Code Coverage [%] for MWTs, RGTs, and SGTs

Project	MWT		RGT		SGT	
	bc	cc	bc	cc	bc	cc
Codec	91.83	95.18	66.06	85.54	84.60	98.73
Coll	76.88	95.91	17.22	66.59	-	-
DBCP	55.42	94.64	15.88	39.29	9.02	52.73
Lang	89.08	100.00	42.64	85.78	-	-
Math	86.00	97.50	13.19	59.39	18.03	37.64
Net	25.88	38.19	13.08	78.39	44.84	97.98
Pool	80.31	100.00	3.50	22.22	13.74	44.00
Vectorz	63.24	95.32	30.06	92.45	64.95	97.78
Average	71.08	89.59	25.20	66.21	39.20	71.48

has a very small number of MWTs). Additionally, in most cases, SGTs achieved higher coverage than RGTs, which is consistent with the results of a recent study [57]. The goal of our study, however, is not to compare test case generation techniques, but rather to evaluate the effect of the test suites they generate on RTS techniques. We discuss the low coverage of RGTs for `Pool` in Section V.

4) Execution Platform: We obtained all data on a (time-shared) cluster in which each node has 32-core 2.3 GHz AMD Opteron Processor 6376 with 32GB of RAM, running Ubuntu 12.04 LTS. We used two versions of Oracle Java: 1.7.0_76 and 1.8.0_31. For each project we used the latest version of Java that could compile it and ran MWTs across all 100 commits we utilized; no project required two versions of Java in the chosen sequence of commits.

D. Threats to Validity

External: The projects used in our study may not be representative. To reduce this threat, we considered projects that vary in size, number of commits, and application domain. We considered a window of 100 commits for each project to limit the machine time and resources required for the experiments, and our results might vary based on the size and location of the window used on the software history.

Our results could also depend on the Randoop and EvoSuite configurations we chose (e.g., test case generation time, maximum numbers of statements per test method, etc.). The version of Randoop we used, however, has more than 50 command line options, and exploring all of its configurations is infeasible. We evaluated Randoop and EvoSuite using their default configurations, which are likely to be used by any novice user and likely reflect what the authors of the tools consider to be appropriate (and generally applicable) configurations.

Internal: Ekstazi, TestTube, Randoop, EvoSuite, and our automation scripts may contain faults, and this could impact our conclusions. We are relatively confident in the correctness of Ekstazi, Randoop, and EvoSuite, because they are robust tools that have been used independently in several prior studies and they have been well maintained. To increase our confidence in our scripts and TestTube (which we implemented for this study), we reviewed the code, tested it on many examples, and manually inspected several results for all projects.

Construct: RTS techniques attempt to reduce the time required to retest a modified program. In practice, such reductions can be measured in terms of the percentages of test cases selected, or in the savings in test execution time (after factoring in the costs of analysis). While the latter metric can more accurately account for savings in cases in which test execution times vary substantially across test cases, in this work we use the former metric, because test execution times can vary unpredictably on the shared cluster that we used in our study. Note, however, that under controlled circumstances, the variance in test execution times on our systems is small, and thus, the former metric is sufficiently accurate.

TABLE IV: Test Selection Ratios for Ekstazi and TestTube

Project	$\mathcal{S}^X$ [%]					
	Ekstazi			TestTube		
	MWT	RGT	SGT	MWT	RGT	SGT
Codec	8.11	29.20	18.95	6.92	22.96	11.95
Coll	13.31	26.18	-	6.39	11.58	-
DBCP	45.63	33.16	33.60	45.44	39.87	26.44
Lang	10.99	19.20	-	5.33	6.64	-
Math	21.84	14.41	33.35	11.83	7.09	11.79
Net	16.29	7.35	9.81	14.26	5.58	4.32
Pool	40.25	42.72	11.22	40.56	49.36	19.51
Vectorz	55.99	52.61	84.37	13.88	9.21	4.08
Average	26.55	28.10	31.89	18.08	19.04	13.01

III. Results

We now present our study results, addressing each of our research questions in turn.

A. RQ1: How Effective are RTS Techniques When Applied to Various Types of Test Suites?

Using our evaluation procedure (Figure 1) we determined, for each commit of each object program, for each RTS technique and type of test suites considered, the number of available test cases and the number of selected test cases. We then computed the selection ratios for each commit. Commits with zero selected tests were *not* included in this computation. This process yielded the data shown in Table IV.

Table IV (left) displays average selection ratios obtained using Ekstazi for all projects, for test cases obtained from MWTs (Column 2), RGTs (Column 3), and SGTs (Column 4). Overall, the selection ratio varies from 8.11% to 55.99% for MWTs, from 7.35% to 52.61% for RGTs, and from 9.81% to 84.37% for SGTs. The highlighted row shows arithmetic means across all projects: $avg(\mathcal{S}^{MWT})$ is 26.55%, $avg(\mathcal{S}^{RGT})$ is 28.10%, and $avg(\mathcal{S}^{SGT})$ is 31.89%.

Columns 5–7 of Table IV show the same selection ratio data for TestTube. TestTube also achieved high selectivity for MWTs and GTs. Specifically, $avg(\mathcal{S}^{MWT})$ is 18.08%, $avg(\mathcal{S}^{RGT})$ is 19.04%, and $avg(\mathcal{S}^{SGT})$ is 13.01%.

On average, TestTube was 46% more effective than Ekstazi (26.55% versus 18.08%) for MWTs, 47% more effective than Ekstazi (28.10% versus 19.04%) for RGTs, and 145% more effective than Ekstazi (31.89% versus 13.01%) for SGTs. (Note, however that Ekstazi can be more efficient than TestTube in terms of execution time [30] and Ekstazi is a safer technique.) While these results for MWTs were not unexpected given our

TABLE V: Correlations (R^2, Spearman's ρ, and Kendall's τ) Between Test Selection Ratios

	Project	$\mathcal{S}^{MWT}$, $\mathcal{S}^{RGT}$			$\mathcal{S}^{MWT}$, $\mathcal{S}^{SGT}$		
		R^2	ρ	τ	R^2	ρ	τ
Ekstazi	Codec	0.83	0.67	0.64	0.93	0.70	0.65
	Coll	0.78	0.93	0.84	-	-	-
	DBCP	0.44	0.70	0.65	0.56	0.88	0.81
	Lang	0.88	0.89	0.79	-	-	-
	Math	0.90	0.90	0.79	0.89	0.75	0.64
	Net	0.60	0.54	0.49	0.77	0.56	0.50
	Pool	0.33	0.47	0.43	0.36	0.90	0.82
	Vectorz	0.97	0.97	0.87	0.71	0.97	0.90
	Average	0.71	0.75	0.68	0.70	0.79	0.72
TestTube	Codec	0.81	0.60	0.57	0.97	0.65	0.62
	Coll	0.80	0.72	0.64	-	-	-
	DBCP	0.42	0.54	0.51	0.41	0.65	0.63
	Lang	0.88	0.79	0.68	-	-	-
	Math	0.73	0.81	0.71	0.25	0.64	0.56
	Net	0.72	0.39	0.37	0.77	0.37	0.33
	Pool	0.33	0.47	0.44	0.31	0.62	0.59
	Vectorz	0.77	0.84	0.71	0.63	0.87	0.75
	Average	0.68	0.64	0.57	0.55	0.63	0.58

prior work [19], [28]–[30], we were not sure what to expect for GTs generated by Randoop and EvoSuite prior to this study, because the effectiveness of Ekstazi and TestTube on the given objects of study depends on the numbers of files and methods that test cases use during execution. More important for the purpose of this work, the results do suggest that we can viably proceed to examine RQ2 and RQ3 relative to this study's data.

B. RQ2: Does the Effectiveness of RTS Techniques Vary Across Types of Test Suites?

Overall, based on the data shown in Table IV, Ekstazi appears to be more effective for MWTs (26.55%) and RGTs (28.10%) than for SGTs (31.89%). On the other hand, TestTube appears to be more effective for SGTs (13.01%) than for MWTs (18.08%) and RGTs (19.04%). This suggests that indeed, RTS techniques are affected differently by the use of different types of test suites.

To further assess the foregoing observations we performed a statistical analysis comparing test selection ratios across types of test suites. For each project and each pair of test suite types, we computed the coefficient of determination (R^2), Spearman's rank correlation coefficient (ρ), and Kendall's correlation coefficient (τ) [38]. Table V shows correlation values between $\mathcal{S}^{MWT}$ and $\mathcal{S}^{RGT}$ (Columns 2–4), and between $\mathcal{S}^{MWT}$ and $\mathcal{S}^{SGT}$ (Columns 5–7). Based on the standard Guilford scale [33], we can assert that the correlation coefficients range from low (< 0.4) to very high (> 0.9). For example, considering the Kendall τ coefficients for Ekstazi comparing $\mathcal{S}^{MWT}$ and $\mathcal{S}^{RGT}$ (top-left portion of the table), two values are moderate (> 0.4) (0.43 for `Pool` and 0.49 for `Net`), two are nearly high (> 0.6) (0.64 for `Codec` and 0.65 for `DBCP`) and four are high (> 0.7); no value is very high (> 0.9). A similar distribution occurs for Ekstazi when comparing $\mathcal{S}^{MWT}$ and $\mathcal{S}^{SGT}$ (top-right portion of the table). Correlation values for TestTube (bottom half of the table) are in most cases lower than values for Ekstazi. This analysis confirms that differences among test suites types can often affect different RTS techniques in different ways.

We conjecture that the differences in the performance of RTS techniques across test suite types may relate to differences in the dependencies per test suite type. To investigate this, we extracted the dependencies for each test case from the data tracked by Ekstazi and TestTube. Table VI provides basic statistics for the numbers of dependencies for MWTs, RGTs, and SGTs. For each project the table shows the maximum numbers of dependencies among all test cases, the mean numbers across all test cases, and the standard deviation.

As the data shows, on most projects MWTs have higher mean numbers of *file dependencies* (tracked by Ekstazi) per test case than RGTs, 24.92 versus 11.18. The mean number of dependencies is also higher for SGTs than MWTs in many instances, 42.78 versus 24.92 on average. Considering the maximum number of dependencies per test case, the order is the same as for the mean values: SGTs (125.67), MWTs (90.12), and RGTs (47.62). We observe very different results for the *method dependencies* tracked by TestTube. MWTs have higher mean numbers of method dependencies (132.99 on average) than RGTs (28.49) and SGTs (27.24). This difference between numbers of class and method dependencies may account for the fact that Ekstazi is more effective for MWTs and RGTs than for SGTs, and TestTube is more effective for SGTs than for MWTs and RGTs.

Table VI also lists the numbers of libraries, i.e., jar files, that each project uses (Column "libs"); these are not compile-time dependencies (defined in `pom.xml`), but rather run-time dependencies tracked by Ekstazi and TestTube. RGTs use smaller numbers of libraries than MWTs (e.g., 2 versus 7 libraries for `DBCP`) and SGTs (e.g., 2 versus 9 for `DBCP`). This finding suggests that an additional metric should be adopted when evaluating tools for automated test case generation: instead of comparing only code coverage (e.g., [57]), future evaluations should also report on coverage of libraries and coverage of code that interacts with libraries.

Finally, from the numbers in Table VI, we can also explain Ekstazi's high $\mathcal{S}$ value for `Vectorz` (84.37%) in the case of SGTs (Table IV). The values in Table VI clearly show (Column "mean") that SGTs depend on many more files (146.47) than MWTs (38.59) and RGTs (23.06). We expect test cases with larger numbers of dependencies to be more frequently selected. This suggests that it might be beneficial to develop a new multi-objective fitness function for EvoSuite that favors test cases with smaller numbers of dependencies and higher overall code coverage.

C. RQ3: Do Differences in RTS Technique Performance Across Types of Test Suites Vary Across Commits?

To answer this question, we analyzed commits when test selection ratios had zero values, i.e., $\mathcal{S}^X = 0$. These cases indicate one of the following scenarios: (1) project changes occurred in files unrelated to the code (such as README files), (2) changes in dependencies did not affect the behavior

TABLE VI: Statistics for Test Case Dependencies

	Project	MWT				RGT				SGT			
		max	mean	sd	libs	max	mean	sd	libs	max	mean	sd	libs
Ekstazi	Codec	31	8.54	5.17	1	35	9.20	5.67	0	36	13.05	6.54	0
	Coll	95	23.03	18.18	1	76	25.36	17.72	0	-	-	-	-
	DBCP	85	44.07	25.41	7	18	2.64	1.17	2	228	32.84	53.42	9
	Lang	51	11.60	10.79	2	45	7.19	5.76	0	-	-	-	-
	Math	102	30.91	21.36	0	97	10.38	10.96	0	154	40.07	31.51	0
	Net	34	10.26	7.64	0	22	5.56	4.25	0	97	16.58	16.94	0
	Pool	139	32.40	36.94	2	12	6.03	2.28	0	23	7.65	3.15	0
	Vectorz	184	38.59	35.05	3	76	23.06	13.48	2	216	146.47	34.32	4
	Average	90.12	24.92	29.31	2.00	47.62	11.18	7.66	0.50	125.67	42.78	24.31	2.17
TestTube	Codec	115	39.75	28.86	1	122	26.30	19.75	0	97	21.78	12.54	0
	Coll	541	161.78	142.78	1	172	49.17	35.82	0	-	-	-	-
	DBCP	553	266.93	181.33	7	72	13.63	9.66	2	176	25.07	25.73	9
	Lang	370	61.10	56.04	2	170	22.76	22.81	0	-	-	-	-
	Math	438	104.29	87.63	0	207	21.16	24.01	0	263	32.78	23.29	0
	Net	126	42.60	28.92	0	58	12.38	9.39	0	68	20.79	10.16	0
	Pool	601	160.65	173.51	2	44	17.97	6.10	0	35	17.94	4.89	0
	Vectorz	2474	226.84	418.31	3	267	64.57	45.84	2	221	45.10	27.06	4
	Average	652.25	132.99	185.28	2.00	139.00	28.49	21.67	0.50	143.33	27.24	17.28	2.17

TABLE VII: Frequencies at Which No Test Cases are Selected

Project	$\mathcal{S}^X = 0$ [%]					
	Ekstazi			TestTube		
	MWT	RGT	SGT	MWT	RGT	SGT
Codec	64.00	87.00	86.00	72.00	90.00	89.00
Coll	45.00	52.00	-	58.00	75.00	-
DBCP	80.00	89.00	84.00	83.00	95.00	93.00
Lang	43.00	58.00	-	52.00	68.00	-
Math	40.00	50.00	62.00	52.00	65.00	71.00
Net	61.00	62.00	62.00	73.00	77.00	63.00
Pool	62.00	90.00	70.00	73.00	94.00	90.00
Vectorz	21.00	30.00	28.00	39.00	52.00	51.00
Average	50.93	64.16	64.19	61.84	76.73	75.67

of test cases (such as changes in source files that affect only debug information in executable files, which Ekstazi ignores), or (3) no test case depended on the changed files. While $\mathcal{S}^X = 0$ is desirable in the first two cases, the third case suggests that test suite quality is sub-optimal, given that no test cases exercise the modified files. Therefore, the results of RTS can be used as a first approximation of test suite quality, and this approximation provides a view on RQ3.

Table VII shows the percentage of commits with $\mathcal{S}^X = 0$ across all projects, for both RTS techniques and all three types of test suites. As the data shows, the percentages are relatively high in most cases. On average, MWTs, RGTs, and SGTs were not selected with the following frequencies: 50.93%, 64.16%, and 64.19% for Ekstazi, and 61.84%, 76.73%, and 75.67% for TestTube. More important in the context of RQ3, the percentage of commits with $\mathcal{S}^X = 0$ is always higher for RGTs and SGTs than for MWTs, which means that GTs were less frequently affected by the set of changes present in the projects we study. In other words, project changes affected MWTs more frequently than they affected RGTs and SGTs.

To obtain additional information on this issue we further inspected our data to determine whether modified files should be covered by test cases. It would be non-trivial to manually inspect 800 commits, so we compared the commits for which

TABLE VIII: Percentage of Commits When Two Types of Test Cases are Mutually Exclusive, i.e., When Ekstazi Selects Test Cases From Only One of the Two Types

Project	$\mathcal{S}^X = 0 \oplus \mathcal{S}^Y = 0$ [%]					
	MWT, RGT		MWT, SGT		RGT, SGT	
Codec	0.00	23.00	0.00	22.00	1.00	0.00
Coll	0.00	7.00	-	-	-	-
DBCP	0.00	12.86	0.00	5.71	7.14	0.00
Lang	0.00	15.00	-	-	-	-
Math	0.00	10.00	1.00	23.00	3.00	15.00
Net	14.00	15.00	14.00	15.00	0.00	0.00
Pool	0.00	28.00	0.00	8.00	21.00	1.00
Vectorz	0.00	9.00	0.00	7.00	5.00	3.00
Average	1.75	14.98	2.50	13.45	6.19	3.17

the condition $\mathcal{S}^X = 0 \oplus \mathcal{S}^Y \neq 0$ holds. These are commits for which we know that some test cases of one type are selected and no test cases of another type are selected. Table VIII shows the percentage of commits for which test cases of only one type are selected. (We show only the results for Ekstazi due to space constraints.)

For several projects, MWTs and GTs complement each other with respect to the results of this analysis, i.e., RTS techniques selected test cases from *only* one type of test suites for several commits. In all but two cases (`Net` and `Math`), Ekstazi selected at least one MWT whenever it selected one of the GTs. On the other hand, there were many cases in which only MWTs were selected. `Net` is an extreme instance in which Ekstazi selected one or more test cases from only one type of test suite in about 30% of commits (14.00% + 15.00%). In conclusion, we can say that in the cases we considered, MWTs commonly subsumed GTs with respect to test selection. We also observed differences between RTS techniques for RGTs and SGTs. Developers may benefit from including both types of test cases in their regression test suites, and researchers may wish to explore closer integration of the two test case generation techniques.

TABLE IX: Code Coverage [%] for RGTs

Project	RGT^b		RGT^c		RGT^d	
	bc	cc	bc	cc	bc	cc
Codec	64.06	85.54	68.41	85.54	61.32	85.54
Coll	22.71	72.84	30.61	80.60	32.16	80.17
DBCP	15.31	35.71	15.75	39.29	16.52	39.29
Lang	44.06	84.86	49.33	88.07	47.74	85.78
Math	11.76	59.04	13.68	51.42	22.96	66.21
Net	16.88	83.92	13.00	79.40	10.67	77.89
Pool	3.50	22.22	3.50	22.22	3.38	22.22
Vectorz	32.51	93.17	35.94	94.96	29.39	90.65
Average	26.35	67.16	28.78	67.69	28.02	68.47

TABLE X: Test Selection Ratios for Ekstazi and TestTube

Project	$\mathcal{S}^X$ [%]					
	Ekstazi			TestTube		
	RGT^b	RGT^c	RGT^d	RGT^b	RGT^c	RGT^d
Codec	32.85	26.97	28.13	27.55	21.14	21.47
Coll	29.06	19.70	18.48	9.37	7.75	8.61
DBCP	33.51	31.54	32.63	40.43	37.47	39.76
Lang	19.58	15.01	16.24	6.15	4.91	6.42
Math	13.91	14.09	14.14	6.39	6.80	5.91
Net	9.84	7.91	7.65	6.00	5.52	5.78
Pool	37.41	34.79	41.07	43.08	37.35	47.60
Vectorz	52.68	48.97	54.62	8.22	5.57	7.48
Average	28.60	24.87	26.62	18.40	15.81	17.88

IV. Implications

Inspired by our confirmation that the effectiveness of RTS techniques can differ for various types of test suites, we wished to explore whether it may be possible to tune a test case generation technique to create test suites with higher code coverage that also yield lower test selection ratios and what is the impact of combining MWTs and GTs. Our exploration involved three steps. First, we explored the impact of Randoop configurations on coverage and test selection ratios. Second, we explored a method for grouping generated test cases that can be integrated with test case generation processes, with the goal of speeding up RTS techniques. Third, we explored the effect of combining test suites of different types.

A. The Impact of Randoop Configurations on RTS Techniques

We repeated our study with test cases generated by feedback-directed techniques using three additional tool configurations supported by Randoop, as shown in the table to the right. We chose to modify tool options that we believe are more likely to lead to test cases of different natures. (Note that both "small-tests" and "alias-ratio" are options for "varying the nature of generated test cases", based on the Randoop documentation [52].) Table II (three rightmost columns), provides statistics on the numbers of generated test cases; code coverage and results for test selection ratios are shown in Tables IX and X.

Config.	Option: randomseed	small-tests	alias-ratio
RGT	✗	✗	✗
RGT^b	✓	✗	✗
RGT^c	✗	✓	✗
RGT^d	✗	✗	✓

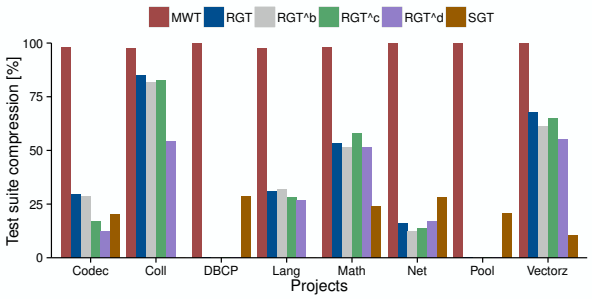

Fig. 2: Test reduction that can be achieved by grouping test cases based on Ekstazi dependencies

Our results show that a configuration that favors shorter test cases (RGT^c) generated test cases that achieved *competitive coverage*, yet these test cases were *less frequently selected* by RTS techniques than test cases generated for other configurations. For example, the last rows in Tables III and IX show that the average class coverage for RGT^c was 67.69%, which is almost identical to the class coverage for RGT (66.21%), RGT^b (67.16%), and RGT^d (68.47%). At the same time, the last rows in Tables IV and X show that Ekstazi's test selection ratio for RGT^c was 24.87% as opposed to 28.10% (RGT), 28.60% (RGT^b), and 26.62% (RGT^d).

B. Test Grouping

We observed that many generated test cases (in each test suite) had the same sets of dependencies, regardless of the test case generation tool configuration used. We computed the numbers of *test clusters* in test suites based on the Ekstazi dependencies; all test cases in the same cluster have exactly the same set of dependencies. (Note that each test case depends on itself and we exclude such dependencies during comparison.) If we merge all test cases with the same dependencies into a single test class, we can reduce the overhead that Ekstazi imposes on the test selection process. This test grouping could be performed as a post-generation phase when the tests are run for the first time (and the grouping can be updated after every `n` project versions, which can be specified by the user) or even integrated within the test case generation tool.

Figure 2 shows the ratio of the number of clusters to the total number of test cases; smaller values indicate higher reductions in the numbers of test classes and larger savings in selection time for Ekstazi. `DBCP` and `Pool` are extremes for RGTs; this was expected as we know that these projects have test suites that achieve low coverage (Table III). While we do not expect that developers would want to group their MWTs based on Ekstazi dependencies, we show the results for MWTs to facilitate comparison. It is clear from these results that MWTs are much more diverse in terms of Ekstazi dependencies, as the reduction is very low.

C. Combining MWTs and GTs

Prior work has suggested that different types of test suites differ in terms of the types of faults they can detect [57],

TABLE XI: Code Coverage for Combinations of MWTs and GTs

Project	MWT		RGT		SGT		MWT+RGT		MWT+SGT		RGT^{all}		MWT+RGT^{all}	
	bc	cc	bc	cc	bc	cc	bc	cc	bc	cc	bc	cc	bc	cc
Codec	91.83	95.18	66.06	85.54	84.60	98.73	93.35	97.59	95.21	98.73	71.74	85.54	94.33	97.59
Coll	76.88	95.91	17.22	66.59	-	-	78.58	96.55	-	-	38.33	83.41	80.72	96.98
DBCP	55.42	94.64	15.88	39.29	9.02	52.73	61.67	98.21	55.25	98.18	17.09	39.29	62.24	98.21
Lang	89.08	100.00	42.64	85.78	-	-	90.56	100.00	-	-	58.70	89.91	91.54	100.00
Math	86.00	97.50	13.19	59.39	18.03	37.64	86.57	97.84	86.00	97.50	31.57	79.75	87.60	98.86
Net	25.88	38.19	13.08	78.39	44.84	97.98	32.87	84.92	55.35	97.98	17.83	83.92	35.88	87.44
Pool	80.31	100.00	3.50	22.22	13.74	44.00	80.31	100.00	81.19	100.00	3.74	22.22	80.56	100.00
Vectorz	63.24	95.32	30.06	92.45	64.95	97.78	67.81	97.12	76.76	98.89	54.80	97.12	75.05	97.48
Average	71.08	89.59	25.20	66.21	39.20	71.48	73.96	96.53	74.96	98.55	36.72	72.64	75.99	97.07

TABLE XII: Test Selection Ratios Obtained by Ekstazi for Combinations of MWTs and GTs

Project	$\mathcal{S}^X$ [%]						
	MWT	RGT	SGT	MWT+RGT	MWT+SGT	RGT^{all}	MWT+RGT^{all}
Codec	8.11	29.20	18.95	10.50	7.40	28.31	10.22
Coll	13.31	26.18	-	21.17	-	20.98	18.23
DBCP	45.63	33.16	33.60	18.29	29.83	32.58	17.93
Lang	10.99	19.20	-	14.01	-	16.88	12.43
Math	21.84	14.41	33.35	14.53	20.88	12.87	12.35
Net	16.29	7.35	9.81	5.45	7.12	8.73	6.28
Pool	40.25	42.72	11.22	11.31	12.15	38.36	10.11
Vectorz	55.99	52.61	84.37	46.95	76.73	49.59	44.03
Average	26.55	28.10	31.89	17.78	25.68	26.04	16.45

rendering combinations of test types potentially useful. For this reason, we also investigated what might happen if GTs were included in regression test suites along with MWTs, as well as if multiple GTs were combined. Table XI shows the coverage achieved for several such combinations; we omitted others due to space limitations. Note that RGT^{all} denotes a test suite that is a union of RGT, RGT^b, RGT^c, and RGT^d. The first three columns in Table XI are the same as those in Table III; we repeat these here to facilitate comparison. Similarly, the first three columns in Table XII are the same as those in Table IV.

The data shows that GTs improve coverage over MWTs by only a few percentage points (e.g., compare the "MWT" column with the "MWT+RGT" and "MWT+SGT" columns). The largest increase in coverage is for `Net` and `DBCP`, which have the lowest (branch) coverage for MWTs. Table XII shows Ekstazi's $\mathcal{S}$ for various combinations of test suites. A large decrease in $\mathcal{S}$ for the combination of MWTs and GTs (compared to selection ratio for GTs) occurs because GTs dominate MWTs (in terms of the number of test cases) and there are many commits for which only MWTs are selected (Table VII). The values of $\mathcal{S}$ for the combination of GTs are similar to the values of $\mathcal{S}$ for individual GTs because these test suites are of similar size and have similar average $\mathcal{S}$ values. In summary, given our current data, we cannot conclude that combinations of types of test suites provide substantial improvements for RTS techniques.

V. Additional Discussion

We now discuss two additional aspects of our results.

One Test Method per Test Class: As noted in Section II, we modified the Randoop configuration to generate one test method per test class, and we post-processed test cases generated by EvoSuite to obtain one test method per test class. By default, Randoop includes up to 500 test methods in each test class and EvoSuite includes all test methods related to one class under test in a single test class. Using one test method per test class was necessary for two reasons. First, we wished to avoid removing excessive numbers of test cases due to compilation errors. Recall from Section II-C that we compile all RGTs across 100 commits and use only test cases that compile at all commits. Therefore, if we had 500 test methods in a single test class, we would remove that test class even if a single test method does not compile for a single commit. Second, Ekstazi and TestTube track dependencies per test class (known as selection granularity [30]) rather than per test method.[3] Although developers often group manually written test methods with similar dependencies into a single test class [30], we have no such expectation from automatically generated test cases, so we split them in separate test classes. This is probably the first change that any developer should make when using Randoop or EvoSuite with Ekstazi. Without this change, considering the number of automatically generated test cases, we would end up with just a few test classes, most of which would be selected at most commits.

Low Coverage: Despite the large number of generated test cases, the coverage achieved for `Pool` and `DBCP` was rather low. In Section II, we noted that these objects have a large number of getter and setter methods, which is the reason for

[3]*Selection granularity* is different from *dependency granularity*: Ekstazi uses file dependency granularity (i.e., tracks accessed files) and TestTube uses method dependency granularity (i.e., tracks used methods), but Ekstazi and our implementation of TestTube use test class selection granularity (i.e., track dependencies for each test class separate).

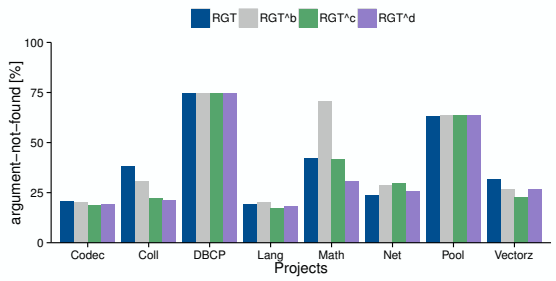

Fig. 3: Percentage of randomly chosen methods for which Randoop could not find the required arguments

the large number of test cases. We investigated the reason behind the low coverage achieved for these two objects of study (Table III). We used Randoop's logs to count the number of randomly chosen methods for which Randoop could not find appropriate arguments. Figure 3 shows, for each object and Randoop configuration, the ratio of unsuccessfully chosen methods to the total number of chosen methods. The ratios for `Pool` and `DBCP` are among the highest. For these objects, Randoop frequently rejected a method invocation because it could not find arguments of appropriate types. Having a test case generation tool monitor and report this ratio could indicate, to a developer, cases in which to expect low coverage and a low test selection ratio. In such cases, the developer may want to explore other test case generation techniques.

VI. Related Work

There has been a lot of work on automated test case generation (e.g., [2], [6]–[8], [17], [27], [31], [40], [49], [51], [54], [58], [59]) and RTS. We have already described Randoop, EvoSuite, Ekstazi, and TestTube, which we used in our study due to the availability of implementations. This section briefly describes other related studies, techniques, and tools. Specifically, we describe (1) several automated test case generation techniques, (2) several RTS techniques, and (3) other work on regression testing related to automated test case generation.

Automated Test Case Generation Techniques: Various techniques and tools have been developed for automated test case generation. Random testing [1], [4], [17] is a naïve approach for generating random sequences of methods. Systematic approaches [6], [12], [31], [58], [59], [67], which exhaustively explore sequences of methods up to a given bound or generate test inputs based on test abstractions, have been used primarily to generate complex data structures. Techniques based on evolutionary algorithms have been shown to be effective [25]–[27], [57], [64]. Other work has explored automatic generation of system test cases from use case specifications [16], [68] and models [18]. We used Randoop and EvoSuite because they are robust tools that are applicable to a wide range of projects and comparable to other approaches [49], [57], [58].

RTS Techniques: Many RTS techniques and tools have been developed over the past three decades. Two recent surveys [22], [72] summarize work on regression testing in general, including RTS. Two other recent surveys [5], [23] focus on contributions related only to RTS. Prior work has proposed techniques that track dependencies for test cases at fine-grained levels (e.g., statements, basic blocks, methods). For example, Chianti [53] tracks dependencies on methods and discovers modified methods by analyzing source code. We used Ekstazi and TestTube in our study to evaluate the effect of test case generation techniques on RTS techniques with coarse-grained and fine-grained dependencies. Ekstazi is the only publicly available RTS tool, and has been shown to be effective in many cases [30]. We implemented TestTube for the purpose of this study by modifying the Ekstazi framework.

Regression Testing with Automated Test Case Generation: There has been relatively little work combining regression testing with automated test case generation. Groce et al. [32] present a test suite reduction technique for use on automatically generated test cases. Their technique is guided by coverage and uses delta debugging [73]. Several researchers [51], [56], [70], [71] have considered the use of automated test case generation in test suite augmentation – the process of improving test suites after code has evolved. None of this work, however, considers RTS techniques.

VII. Conclusions

We have designed and presented the results of the first empirical study to evaluate the effects that different types of test suites have on RTS techniques. Specifically, we considered two types of automatically generated test suites—test suites generated by feedback-directed random techniques and search-based techniques—along with manually written test suites. We assessed the effects of these types of test suites on two RTS techniques: a "fine-grained" technique that tracks dependencies on methods (TestTube) and a "coarse-grained" technique that tracks dependencies on files (Ekstazi). We performed our study on eight open-source projects across 800 commits. (To the best of our knowledge, this is the largest number of commits used in any RTS study.) Our results showed that on average the fine-grained RTS technique was more effective for test suites created by search-based test case generation techniques (13.01% compared to 19.04%), whereas the coarse-grained RTS was more effective for test suites created by feedback-directed random techniques (28.10% compared to 31.89%). We also found that commits affect the performance of RTS techniques, and do so differently for different types of RTS techniques.

Acknowledgments. We thank the fellow students of EE 382V (Software Evolution) at The University of Texas at Austin for constructive discussions on the material presented in this paper. We also thank Marko Dijmasevic, Ahmet Celik, Sarfraz Khurshid, and Marko Vasic for their feedback on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-0845628, CCF-1566363, CNS-1239498 (to UTA) and CCF-1526652 (to UNL), and by a Google Faculty Research Award (to UTA).

REFERENCES

[1] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Transactions on Software Engineering*, pages 258–277, 2012.

[2] A. Aydin, M. Alkhalaf, and T. Bultan. Automated test generation from vulnerability signatures. In *International Conference on Software Testing, Verification and Validation*, pages 193–202, 2014.

[3] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *Transactions on Software Engineering and Methodology*, 10(2):149–183, 2001.

[4] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.

[5] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[7] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering*, pages 443–446, 2008.

[8] T. Y. Chen, F. Kuo, H. Liu, and W. E. Wong. Code coverage of adaptive random testing. *Transactions on Reliability*, 62(1):226–237, 2013.

[9] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: A system for selective regression testing. In *International Conference on Software Engineering*, pages 211–220, 1994.

[10] Commons Codec home page. https://github.com/apache/commons-codec.git.

[11] Commons Collections home page. https://github.com/apache/commons-collections.git.

[12] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *International Symposium on Foundations of Software Engineering*, pages 185–194, 2007.

[13] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Automated Software Engineering*, pages 433–444, 2009.

[14] B. A. Daou. Regression test selection for database applications. *Advanced Topics in Database Research*, 3:141–165, 2004.

[15] Commons DBCP home page. https://github.com/apache/commons-dbcp.git.

[16] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, 2005.

[17] S. Ducasse, M. Oriol, and A. Bergel. Challenges to support automated random testing for dynamically typed languages. In *International Workshop on Smalltalk Technologies*, pages 9:1–9:6, 2011.

[18] G. Edwards, Y. Brun, and N. Medvidovic. Automated analysis and code generation for domain-specific models. In *Joint Working Conference on Software Architecture & European Conference on Software Architecture*, pages 161–170, 2012.

[19] Ekstazi home page. http://www.ekstazi.org.

[20] S. Elbaum, P. Kallakur, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 13(2):65–83, 2001.

[21] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[22] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement*, pages 3–16. Springer-Verlag, 2010.

[23] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information & Software Technology*, 52(1):14–30, 2010.

[24] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *International Symposium on Empirical Software Engineering and Measurement*, pages 22–31, 2008.

[25] EvoSuite home page. http://www.evosuite.org/.

[26] G. Fraser and A. Arcuri. Whole test suite generation. *Transactions on Software Engineering*, 39(2):276–291, 2013.

[27] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis*, pages 147–158, 2010.

[28] M. Gligoric. *Regression Test Selection: Theory and Practice*. PhD thesis, The University of Illinois at Urbana-Champaign, 2015.

[29] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *International Conference on Software Engineering, Demo*, pages 713–716, 2015.

[30] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[31] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234, 2010.

[32] A. Groce, A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *International Conference on Software Testing, Verification and Validation*, pages 243–252, 2014.

[33] J. P. Guilford. *Fundamental Statistics in Pyschology and Education*. McGraw-Hill, 1956.

[34] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*, pages 223–233, 2015.

[35] J. Hartmann. Applying selective revalidation techniques at Microsoft. In *Pacific NW Software Quality Conference*, pages 255–265, 2007.

[36] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *International Conference on Software Engineering*, pages 483–493, 2015.

[37] JaCoCo home page. http://eclemma.org/jacoco/.

[38] M. Kendall. A new measure of rank correlation. *Biometrika*, 1(2):81–89, 1938.

[39] Commons Lang home page. https://github.com/apache/commons-lang.git.

[40] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t). In *Automated Software Engineering*, pages 494–505, 2015.

[41] Commons Math home page. https://github.com/apache/commons-math.git.

[42] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *Transactions on Software Engineering and Methodology*, 18(2):4:1–4:36, 2008.

[43] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *International Symposium on Foundations of Software Engineering*, pages 135–144, 2007.

[44] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *International Conference on Software Testing, Verification and Validation*, pages 21–30, 2011.

[45] Commons Net home page. https://github.com/apache/commons-net.git.

[46] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering*, pages 117–132, 2014.

[47] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

[48] C. Pacheco. *Directed Random Testing*. PhD thesis, Massachusetts Institute of Technology, 2009.

[49] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[50] Commons Pool home page. https://github.com/apache/commons-pool.git.

[51] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Automated Software Engineering*, pages 397–406, 2010.

[52] Randoop manual. http://randoop.googlecode.com/hg-history/v1.3.3/doc/index.html.

[53] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.

[54] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Automated Software Engineering*, pages 23–32, 2011.

[55] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *Transactions on Software Engineering*, 22(8):529–551, 1996.

[56] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Automated Software Engineering*, pages 218–227, 2008.

[57] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Automated Software Engineering*, pages 201–211, 2015.

[58] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering*, pages 262–277, 2011.

[59] R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov. A comparison of constraint-based and sequence-based generation of complex input data structures. In *Workshop on Constraints in Software Testing, Verification and Analysis*, pages 337–342, 2010.

[60] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *International Symposium on Foundations of Software Engineering*, pages 246–256, 2014.

[61] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *International Symposium on Software Testing and Analysis*, pages 97–106, 2002.

[62] Streamline testing process with test impact analysis. http://msdn.microsoft.com/en-us/library/ff576128%28v=vs.100%29.aspx.

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

[64] P. Tonella. Evolutionary testing of classes. In *International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[65] Tools for continuous integration at Google scale. http://www.youtube.com/watch?v=b52aXZ2yi08.

[66] Vectorz home page. https://github.com/mikera/vectorz.git.

[67] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48, 2006.

[68] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal. Automatic generation of system test cases from use case specifications. In *International Symposium on Software Testing and Analysis*, pages 385–396, 2015.

[69] D. Willmor and S. M. Embury. A safe regression test selection technique for database driven applications. In *International Conference on Software Maintenance*, pages 421–430, 2005.

[70] Z. Xu, Y. Kim, M. Kim, M. B. Cohen, and G. Rothermel. Directed test suite augmentation: an empirical investigation. *Journal of Software Testing, Verification and Reliability*, 25(2):77–114, 2015.

[71] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *International Symposium on Foundations of Software Engineering*, pages 257–266, 2010.

[72] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[73] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Transactions on Software Engineering*, 28(2):183–200, 2002.

[74] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance*, pages 23–32, 2011.

[75] J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *International Symposium on Software Reliability Engineering*, pages 225–234, 2005.