A Framework for Checking Regression Test Selection Tools

Chenguang Zhu[1], Owolabi Legunsen[2], August Shi[2], and Milos Gligoric[1]
[1]The University of Texas at Austin, [2]University of Illinois at Urbana-Champaign
Email: cgzhu@utexas.edu, legunse2@illinois.edu, awshi2@illinois.edu, gligoric@utexas.edu

***Abstract*—Regression test selection (RTS) reduces regression testing costs by re-running only tests that can change behavior due to code changes. Researchers and large software organizations recently developed and adopted several RTS tools to deal with the rapidly growing costs of regression testing. As RTS tools gain adoption, it becomes critical to check that they are correct and efficient. Unfortunately, checking RTS tools currently relies solely on limited tests that RTS tool developers manually write.**

We present RTSCheck, the first framework for checking RTS tools. RTSCheck feeds evolving programs (i.e., sequences of program revisions) to an RTS tool and checks the output against rules inspired by existing RTS test suites. Violations of these rules are likely due to deviations from expected RTS tool behavior, and indicative of bugs in the tool. RTSCheck uses three components to obtain evolving programs: (1) AutoEP automatically generates evolving programs and corresponding tests, (2) DefectsEP uses buggy and fixed program revisions from bug databases, and (3) EvoEP uses sequences of program revisions from actual open-source projects' histories. We used RTSCheck to check three recently developed RTS tools for Java: Clover, Ekstazi, and STARTS. RTSCheck discovered 27 bugs in these three tools.

I. Introduction

Regression testing is an important, widely used, but costly approach for checking that code changes do not break previously working functionality. Regression testing costs arise from running tests on each program revision; such costs have been growing quadratically with the growth in the number of tests and with increasing frequency of code changes [1]–[4].

Regression test selection (RTS) [5]–[8] reduces regression testing costs by selecting to re-run, on each new program revision, only a subset of tests that can change behavior due to code changes, i.e., *affected tests*. A typical RTS technique collects dependencies (e.g., methods, classes) for each test and selects to re-run only tests whose dependencies changed. An RTS technique is *safe* if it does not miss to select any affected test and *precise* if it selects only affected tests.

To deal with the rapidly growing costs of regression testing, several RTS tools were recently developed and adopted by both industry and researchers. Industry examples include Microsoft's Test Impact Analysis for .NET [9], which ships with Visual Studio to millions of developers, and Clover's Test Optimization for Java [10], which was recently open-sourced to increase adoption. Researchers also developed several RTS tools in the last five years alone [11]–[14]; some of these tools have been adopted by large software organizations [15].

As RTS tools gain adoption and become more mainstream, it becomes critical and timely to check their *correctness* and *efficiency*. We say an RTS tool is correct if it is safe and precise, subject to the implemented RTS technique. RTS tool efficiency is measured by comparing its end-to-end time (i.e., test selection time plus execution time for selected tests) with RetestAll, i.e., running all tests at each revision.

Unfortunately, there is no systematic approach for checking correctness and efficiency of RTS tools. Checking RTS tools currently depends solely on the limited sets of tests that each RTS tool developer manually writes. Prior research established a framework for analytically evaluating RTS *techniques* [16], and several researchers semi-formally proved safety and computational complexity of RTS techniques [17]–[19]. However, these proofs and analyses may not carry over to the RTS *tools* that implement those techniques; implementing a technique in a tool requires engineering and, like any other software, RTS tools may contain bugs. A framework for checking RTS tools will (1) enable researchers to compare existing RTS tools and check future RTS tools, and (2) provide greater confidence to developers who are considering to adopt RTS tools.

We present RTSCheck, a novel framework for checking RTS tools. RTSCheck feeds *evolving programs*, i.e., sequences of program revisions, to an RTS tool and checks the output against *rules* that specify *likely* violations of expected behavior. RTSCheck currently uses seven hand-crafted rules, inspired by developer-written tests for RTS tools; users can extend the set of rules. RTSCheck detects violations in three categories. (1) RTSCheck detects a *likely safety violation* if an RTS tool does not select expected tests, e.g., not selecting to run newly failed tests that fail in RetestAll. (2) RTSCheck detects a *likely precision violation* if an RTS tool selects unnecessary tests, e.g., running *all* tests the second time when run twice on the same program revision. (3) RTSCheck detects a *generality violation* if an RTS tool does not integrate well with the program, leading to unexpected behavior, e.g., failing more tests than RetestAll due to incorrect instrumentation. RTSCheck also generates an *efficiency report*, which shows if an RTS tool takes longer to run (on average) than RetestAll. RTSCheck uses the common assumptions in RTS research that tests are not flaky [20]–[22], and there is no test-order dependency [23]–[25]. All violations RTSCheck detects are due to *implementation issues* or *limitations of the underlying RTS technique* that were unknown *a priori*; currently, we map violations to these two root-causes manually.

RTSCheck has three *components* for obtaining the evolving programs (i.e., code and tests) for testing RTS tools. First,

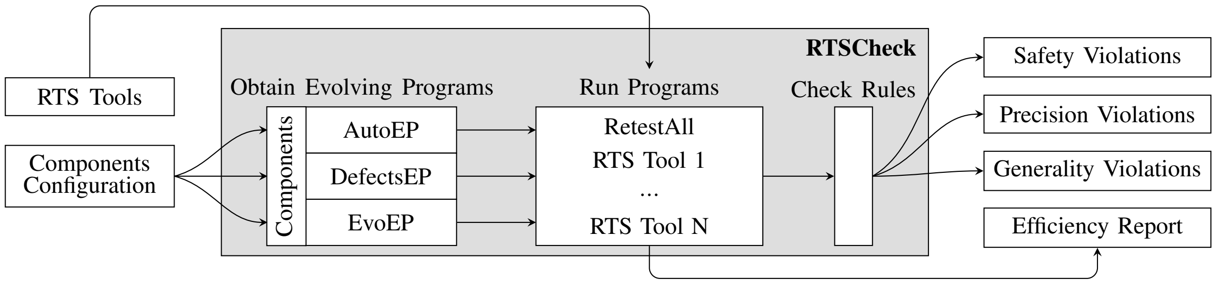

Fig. 1: An overview of the RTSCHECK framework

the *AutoEP* component automatically generates a code revision, randomly generates tests for that revision, and systematically modifies the old revision with a set of evolution operators to obtain subsequent revisions of code and tests. Second, the *DefectsEP* component obtains evolving programs with two revisions from bug databases, using the fixed version as the first revision and the buggy version as the second revision. Finally, the *EvoEP* component obtains evolving programs by extracting program revisions from software repositories.

Our current RTSCHECK implementation supports checking and comparing RTS tools for Java. We used RTSCHECK to check and compare three recent RTS tools: Clover [26], Ekstazi [12], and STARTS [18]. Clover is developed by industry, while Ekstazi and STARTS are developed by researchers. We used RTSCHECK to obtain a total of 31K evolving programs, which have a total of 4M tests. RTSCHECK reported 24K violations; we inspected a subset of these violations (207) and mapped them to 20 implementation issues, seven limitations of the underlying RTS techniques, and two false positives. We reported all 27 bugs to the developers of these RTS tools, four of which were already known to the developers. The developers already confirmed 10 of the 23 previously unknown bugs. Each RTSCHECK component contributed to discovering several unique bugs, and we found at least six bugs in each RTS tool that we checked.

This paper makes the following contributions:

- **Framework**: We present RTSCHECK, the first framework for systematically checking RTS tools for likely violations of expected behavior and for reporting efficiency. We are the first to apply automatic generation of evolving programs and existing bug databases for checking RTS tools.
- **Implementation**: We implement RTSCHECK to check RTS tools for Java. RTSCHECK can be extended to check new RTS tools, support new components for obtaining evolving programs, or use different rules. RTSCHECK is available at http://cozy.ece.utexas.edu/rtscheck.
- **Evaluation**: We deployed RTSCHECK to check three RTS tools: Clover, Ekstazi, STARTS. RTSCHECK discovered 27 bugs, four of which were known. The RTS tool developers so far confirmed 10 of the remaining 23 bugs.

II. THE RTSCHECK FRAMEWORK

Figure 1 shows an overview of RTSCHECK. RTSCHECK takes two inputs: (1) a configuration file for setting up the components that obtain evolving programs, and (2) the RTS tools to check. Based on these inputs, RTSCHECK obtains evolving programs; feeds these programs, one at a time, to the RTS tools; and checks for likely violations (violations for short). We first describe the rules that RTSCHECK uses to detect violations, and then we describe each component for obtaining evolving programs.

TABLE I: Rules for Detecting Violations from Running an Evolving Program with RetestAll and at least One RTS Tool

Id	Violation Description	Type
R1	In some revision, the number of newly failed tests when run with the tool is *lower* than with RetestAll	safety
R2	In some revision, the tool selects zero tests but all other tools select all tests	
R3	In all revisions, the tool selects all tests	precision
R4	In some revision, the tool selects all tests but all other tools select zero tests	
R5	The first two revisions are the same, and the tool selects one or more tests in the second revision	
R6	In the first revision, the tool selects a different number of tests than RetestAll	generality
R7	In some revision, the number of failed tests when run with the tool is *greater* than with RetestAll	

A. Rules for Detecting Violations

We define an *evolving program* as a sequence of n program revisions $(P_0, P_1, ..., P_{n-1})$. Each program revision is a tuple of code under test and a test suite. For a given evolving program, RTSCHECK runs each program revision with the RTS tools being checked; the results of test execution and tool-specific intermediate data are stored as metadata. The metadata is available when running the next program revision and enables RTS tools to perform selection.

Our rules for detecting violations apply to one evolving program at a time; the rules are defined over the metadata available after executing the evolving program. Table I shows our rules for detecting violations. We assume that RTSCHECK executes the given evolving program with RetestAll and at least one RTS tool, although rules R2 and R4 apply only if more than one RTS tool is provided as the input. For each rule, we show a unique Id, a short description of when the rule is violated, and the type of the violation that the rule detects; there are three types of violations: safety violations, precision violations, and generality violations.

1) Safety Violations: Rules R1 and R2 detect safety violations. A safety violation occurs when the RTS tool may be selecting fewer newly failed tests than should be selected; a newly failed test passed in the old revision but fails in the new revision. Violating rule R1 indicates a bug (we assume no flaky tests and no test order dependencies). Not selecting a newly failed test means that an RTS tool could cause users to miss a fault in the code. Violating rule R2 may not indicate a true bug, because it compares an RTS tool against other tools. However, in such cases, the difference in selection (selecting no tests versus all other tools selecting all tests) is extreme, so it is highly likely there is a bug in the tool.

2) Precision Violations: Rules R3, R4, and R5 detect precision violations. A precision violation occurs when the RTS tool may be selecting too many tests, more than are necessary. Of these rules, there is no guarantee that violating rules R3 and R4 indicate actual bugs in the RTS tool; it may be the correct behavior for the RTS tool to be selecting all the tests as they may all be affected tests. However, again, such scenarios are rather extreme and likely indicative of a bug where all other tools select no tests, or if on every revision a tool selects all tests. Violating rule R5 indicates a true bug: if there is no change, there should be no affected tests.

3) Generality Violations: Rules R6 and R7 detect generality violations. A generality violation occurs when using the RTS tool with the code and tests leads to different behavior than what is expected, such as crashing or failing more tests than with RetestAll. Violating either of the two rules indicates a true bug in the RTS tool. For rule R6, an RTS tool on a fresh, first revision should always select all the tests, as RetestAll does. Not selecting all the tests indicates the tool is not working properly with the code/tests. While violating R6 means running fewer tests than should be run, it is not a safety violation, as there is no *change* that leads to any affected tests. For rule R7, if an RTS tool results in more failed tests than RetestAll, the extra failed tests must be due to bad integration with the tool, e.g, the tool caused tests to behave differently, or the tool is crashing for some tests.

Our rules are inspired by the assertions from existing manually written tests for RTS tools, capturing common expected behaviors of RTS tools. Note that while assertions from manually-written tests helped design the rules, RTS tool developers usually check these assertions on very few, if any, evolving programs. Starting with some of the base assertions from existing tests, we modified them to only test extreme cases. For example, instead of having a rule that is violated when an RTS tool selects fewer tests than other RTS tools, our rule R2 is a more extreme version of this rule. Intuitively, having more extreme rules leads to fewer false positives. The rules we use here do not necessarily find all bugs in RTS tools. However, RTSCHECK has a modular design and provides a way to extend the set of rules that can help detect more violations that lead to finding more bugs. We plan to study various extensions in the future.

B. The AutoEP Component

AutoEP obtains evolving programs via automated code generation, test generation and code evolution. Our key idea is to apply *bounded exhaustive testing* with randomly generated tests and state comparison for checking RTS tools. Additionally, we develop a novel set of program evolution operators.

AutoEP works in three main steps: (1) generate the first revision in an evolving program, (2) generate tests for the first-revision programs, and (3) evolve those first-revision programs to obtain corresponding second-revision programs. A program may evolve in multiple ways, so AutoEP obtains multiple evolving programs from a first-revision program.

1) Program Generator: AutoEP uses JDolly [27] to generate the first-revision programs. JDolly systematically generates Java programs up to specified bounds and was originally developed for testing Java refactoring engines [28]. We choose JDolly because it exhaustively generates programs with complex relations among code elements (e.g., class inheritance). Table II shows the user-specifiable constraints we use to tune program generation for JDolly. The table shows a unique identifier for each constraint (Id) and a short summary of each constraint (Summary (see [27])). More details about the constraints are available elsewhere [29].

TABLE II: Program Generation Constraints Used in AutoEP

Id	Summary (see [27])
JD1	NoConstraints
JD2	ClzWithMethodAndSuperClz
JD3	ClzWithMethodAndSubClz
JD4	ClzWithFieldAndSuperClz
JD5	ClzWithMethodAndField
JD6	SomeInheritance
JD7	SomeMethod
JD8	SomeField
JD9	SomeCaller
JD10	SomeFieldSomeFieldAccess

Note that programs generated by JDolly may not compile. Therefore, AutoEP contains a post-processing step to remove programs that do not compile.

2) Test Generator: AutoEP uses the Randoop tool [30], [31] to generate tests. Each test method generated by Randoop is a sequence of method calls. The method sequence length and the number of test methods to generate can be specified as AutoEP inputs; we evaluate with maximum sequence lengths of {1, 2, 4, 100} and (up to) 50 tests per length. The other input to Randoop is the set of classes for which to generate tests. The list of classes we provide to Randoop are the first-revision classes generated by JDolly. Randoop starts with an empty set of sequences and randomly chooses, in each step, a method to invoke from one of the first-revision classes. The call sequence is extended until the specified limit is reached or invoking the current sequence causes an exception so that further extension of the sequence is not beneficial. The arguments for each method call in the sequence are selected either from a predefined pool for each type (e.g., `null` for reference types) or from the results of prior method calls in the same method call sequence.

Oracles in Randoop tests are limited, so we additionally capture *program state* as new oracles for the tests. Specifically, the program state contains objects of all classes in the generated program, including values of all primitive (inherited) fields and information about their type (i.e., the class hierarchy).

We capture the program state at the end of a test run by a lightweight heap traversal. We treat the state captured at the end of a test run in the first-revision program as the expected value of the state. Essentially, one can manually create an assertion that fails if the state captured at the end of a test run does not match this expected value of the state. If running the test later, e.g., after evolution, the state does not match this expected state, then this assertion fails. Dynamic instrumentation that we use to capture program state is lightweight: it only adds a single line to each constructor. We confirmed on a large number of examples that our instrumentation does not conflict with that used by the dynamic RTS tools in our study, nor do they impact dependencies collected by the RTS tools.

3) Program Evolver: Automatically checking RTS tools requires at least two revisions of an evolving program, which is unlike prior work on program (and test) generation for testing compilers [32]–[36], refactoring engines [27], [37], [38], etc., which generate many *single-revision* programs. Our approach for evolving the first revision of an evolving program into the second revision uses mutations, similar in spirit to how the EMI approach [39]–[42] creates program variants for compiler testing. The differences with EMI are the set of program evolution operators used for evolving the initial revision and the goal of generating programs. *The goal of many operators in AutoEP is to change the behavior of the code under test in ways that affect test outcomes.* To generate second-revision programs, AutoEP mutates first-revision programs using a set of *program evolution operators* that we define based on the literature on developing safe RTS techniques for object oriented programming languages [17], [43], [44].

TABLE III: Program Evolution Operators Available in AutoEP

Id	Description
E1	Add extends
E2	Copy field
E3	Copy field and replace
E4	Copy method
E5	Copy method and replace
E6	Evolve to next program
E7	Increase constants
E8	Remove extends
E9	Remove method

Table III shows the program evolution operators supported in AutoEP. The first column shows Id for each operator that will be used later in this document, and the second column briefly describes each operator. "Add extends" (E1) adds an `extends` keyword to a class and systematically chooses a superclass. "Copy field" (E2) copies a field from one class to another. "Copy field and replace" (E3) copies a field from one class to another and changes its initial value (if primitive). "Copy method" (E4) copies a method from one class to another. "Copy method and replace" (E5) copies a method from one class to another and changes a constant in its body. "Evolve to next program" (E6) evolves a program to a subsequent program, considering the order in which the programs were generated. "Increase constant" (E7) replaces a constant with a larger value. "Remove extends" (E8) removes an `extends` keyword. "Remove method" (E9) removes a method.

Operators E1, E2, E3, E4, E5, E8, and E9 impact class relationships. E6 corresponds to a random program evolution. Finally, E7 modifies various code elements where constants can appear, e.g., field initialization or return statement. Each operator may be applied to several locations in a first-revision program, resulting in many evolving programs. AutoEP's operators are related to operators in mutation testing [45]–[49], which are used to evaluate test-suite quality. For example, "Increase constant" is available in most mutation testing tools. However, other AutoEP operators have no equivalent previously-proposed mutation operator.

C. The DefectsEP Component

The DefectsEP component obtains evolving programs by extracting fixed and buggy revisions from bug databases. Our goal is to use a real bug-introducing change and a failing test for checking RTS tools. Several bug databases follow a similar structure: there are two program revisions for each bug, one revision corresponding to the buggy program revision and the other corresponding to the fixed program revision. These programs are usually (large) open-source projects and the bugs are actual bugs fixed by developers of those projects. There is usually at least one test that fails in the buggy revision (i.e., the bug-revealing test) and passes in the fixed revision.

The main idea behind DefectsEP is to reverse the order of the buggy and fixed revisions to simulate a program change that leads to failing tests. If an RTS tool is integrated in such an evolving program, the tool should always select the failing test(s). Although our original motivation is to check for safety violations, we still check all applicable rules. For the case of rule R5, because there cannot be two revisions that are the same (otherwise, there cannot be a buggy and a fixed revision), we also run the fixed revision twice to have the first two revisions be the same and see if R5 is violated.

In our current version of RTSCHECK, the DefectsEP uses the Defects4J bug database [50]. Defects4J includes a large number of bugs and it has been used for many software engineering research tasks [51]–[53]. We are the first to use Defects4J for evaluating RTS tools.

D. The EvoEP Component

The EvoEP component obtains an evolving program by extracting program revisions from existing software repositories. Furthermore, like with DefectsEP, we run the first revision twice to ensure the first two revisions are the same, which helps us check if R5 gets violated. The main motivation for having EvoEP is to evaluate RTS tools with evolving programs with more than two revisions (the previous two components use only two revisions). Thus, EvoEP may potentially discover bugs that require more than two revisions to expose. Moreover, checking an RTS tool with EvoEP can be seen as integration testing. EvoEP extracts evolving programs from projects with complex setups, so it also has the potential to discover bugs that manifest only with specific program configurations. Additionally, the best way to evaluate efficiency is likely by observing execution time on longer-running evolving programs. Finally, is is important to check RTS tools on actual program changes over a period of time.

TABLE IV: Number of Generated Programs (Base) and Number of Generated Evolving Programs (#G - Total and #C - Compilable) for Various Modes using AutoEP

	Base	E1		E2		E3		E4		E5		E6		E7		E8		E9	
		#G	#C	#G	#C	#G	#C	#G	#C	#G	#C	#G	#C	#G	#C	#G	#C	#G	#C
JD1	88	264	0	352	352	0	0	528	58	528	58	88	29	264	264	176	0	264	0
JD2	390	1404	159	0	0	0	0	3120	1950	3120	1950	390	230	1074	1074	702	293	1560	293
JD3	377	1317	158	0	0	0	0	3016	2421	3016	2421	377	240	1039	1039	692	109	1508	273
JD4	199	512	512	199	199	199	199	85	85	85	85	199	199	199	199	142	142	85	85
JD5	85	224	26	0	0	0	0	60	47	60	47	85	57	60	60	58	42	60	0
JD6	399	1374	101	0	0	0	0	2394	1873	2394	1873	399	230	1066	1066	739	289	1197	277
JD7	399	1341	58	0	0	0	0	2349	2043	2349	2043	399	247	1197	1197	730	84	1197	0
JD8	251	885	793	1004	1004	1004	1004	502	459	502	459	251	226	502	502	458	420	251	223
JD9	365	1248	153	0	0	0	0	2190	1826	2190	1826	365	189	826	826	679	174	1095	80
JD10	80	280	115	158	158	158	158	474	370	474	370	80	39	160	160	144	73	240	96

III. Experiment Setup

In this section, we describe the RTS tools used in the evaluation and present evolving programs obtained by RTSCheck. We make a replication package for our evaluation publicly available on our website [54].

A. The RTS Tools Under Evaluation

We use three RTS tools in our evaluation: Clover [26], Ekstazi [12], [55] and STARTS [18], [56]. All the tools work for Java. Further, Ekstazi and STARTS are developed by researchers. Clover was developed in industry and started life as a proprietary product but is now open-source.

Clover. In Clover [26], the Test Optimization feature [10] performs RTS. Clover performs *source-code instrumentation* prior to compilation. Then, at runtime, it records in a database a mapping from each method under test to the set of test methods that use the method under test. After a change, Clover first finds the methods under tests that changed, then it queries its database to find which tests used the changed methods. Clover re-instruments the files that contain changed methods and keeps the same instrumentation for unchanged files. Clover re-runs tests that it found to use changed method(s) from the previous revision, plus any test(s) not already in its database, e.g., newly added tests. Finally, after running the tests, Clover updates the method-to-tests mapping in its database with information from the current run, in preparation for a future run. We use Clover 4.2.0 and the default configuration.

Ekstazi. Ekstazi [12], [55] uses *dynamic binary instrumentation* to track the class dependencies of test classes. More specifically, Ekstazi tracks, as test dependencies, the classes (i.e., underlying compiled `.class` files) that are used while executing each test class. Ekstazi computes and stores a checksum for each test dependency in a revision. Then, after a code change, Ekstazi recomputes the checksum of all the test dependencies to see which ones have changed. The affected tests computed by Ekstazi are all test classes for which at least one dependency has a different checksum in the previous and current revision, plus any newly added test(s). Finally, while running the affected tests in the current revision, Ekstazi uses its instrumentation to track and update the dependencies of the affected tests, in preparation for a future run. We use Ekstazi 5.1.0 and the default configuration.

STARTS. STARTS [18], [56] *statically* computes the dependencies of each test class and does not require any instrumentation. First, STARTS uses jdeps [57] to extract the dependencies of each class in the application. STARTS computes as test dependencies the reflexive and transitive closure for each node that represents a test class in the dependency graph. Note that (1) STARTS can be imprecise because the dependencies found by jdeps are only potentially used classes and are not necessarily runtime dependencies, (2) the constant pool in a `.class` file contains the list of fully qualified names of all classes that are used in the source file, and (3) STARTS can miss dependencies when the relationship between classes happens only via reflection. STARTS computes changes and computes/stores checksums in the same way as Ekstazi. We use STARTS 1.3 and the default configuration.

B. Evolving Programs

1) AutoEP: Table IV shows the number of generated evolving programs using different AutoEP *modes*, i.e., a combination of a program generation constraint and a program evolution operator. Each row of the table shows the constraints used in program generation, and each column shows one way to evolve those programs.

We configured AutoEP to generate 400 programs (i.e., first revisions of evolving programs) for each mode; we limit the number of programs to make the experiments feasible. Table IV shows, in the "Base" column, the number of those programs (out of 400) that successfully compile. AutoEP evolves only the programs that can be compiled. For each program evolution operator, we show the number of generated evolving programs (#G) and the number of those that can be successfully compiled in the second revision (#C); the following sections use only compilable evolving programs. As we expected, some operators are better than others at generating compilable evolving programs. For example, increasing a constant (E7) or copying a field (E2) does not introduce any compilation error. On the other hand, removing a method (E9) frequently leads to a compilation error, because those methods are invoked from at least one of the tests. Finally, as expected, copying a field or increasing a constant does not create any evolving program when no field is present in the original program (e.g., mode JD2 + E2) or all fields are references (JD1 + E3). In total, AutoEP generated 31,104 evolving programs.

TABLE V: DefectsEP Subjects (#P=Number of Evolving Programs, #VP=Number of Valid Evolving Programs)

Project Name	#P	#VP
JFreechart	26	0
Closure-compiler	133	0
Commons-lang	65	26
Commons-math	106	100
Mockito	38	0
Joda-time	27	26
Total	395	152
Avg.	65.83	25.33

2) DefectsEP: In theory, we could use all examples available in the Defects4J bug database. However, as our evaluation in this paper uses a specific set of RTS tools, we filter out the examples on which it is known that one of the RTS tool does not explicitly support, e.g., using a build system one of the RTS tools does not support. Specifically, we perform the following steps to obtain evolving programs for DefectsEP:

a) Start with all the 395 examples in the Defects4J repository (SHA 6bc92429) [50].
b) Filter out the 72 non-Maven examples; we filter non-Maven projects because Clover and STARTS currently support only the Maven build system.
c) Filter out the 138 examples that cannot build, due to issues such as old dependencies that cannot be found anymore.
d) Filter out the 33 examples that do not compile with Java 8; we use Java 8 because STARTS does not work with earlier Java versions.
e) Exclude the seven tests that are flaky or fail consistently on the fixed program version (one test in Commons-lang, one test in Commons-math, and five tests in Joda-time); we detect flaky tests by running each test three times and observing differences in test outcomes.

Table V shows the name of each project, total number of evolving programs, and the number of valid evolving programs after the aforementioned steps.

TABLE VI: EvoEP Subjects

Project Name	SHA	#Tests
Closure-compiler	8594a5cb	357
DBCP	23f6717c	43
IO	078af456	104
Commons-math	085816b7	483
Net	4e5a6992	43
Graphhopper	14d2d670	141
Guava	34c16162	496
HikariCP	471e27ec	35
OpenTripPlanner	8f1794da	139
Streamlib	6e0edb5f	25
Total	N/A	1,866
Avg.	N/A	186.6

3) EvoEP: The EvoEP component can be configured to extract evolving programs from any project that uses a version control system, such as Git. In our experiments we use projects that are available on GitHub, use the Maven build system, and were recently used in research on regression testing [13], [18]. We limit the max number of revisions to 20 to ensure that running experiments and inspecting violations is feasible.

Table VI shows the list of projects used by EvoEP. For each project, we show its name, the latest SHA used for the experiments, and the number of tests at the latest SHA.

IV. Evaluation

To assess the benefits of using RTSCheck for checking RTS tools, we answer the following research questions:

RQ1: What safety violations and bugs are detected by RTSCheck, and which components provide evolving programs on which violations are detected?

RQ2: What precision violations and bugs are detected by RTSCheck, and which components provide evolving programs on which violations are detected?

RQ3: What generality violations and bugs are detected by RTSCheck, and which components provide evolving programs on which violations are detected?

RQ4: What can be learned about efficiency of RTS tools by using various RTSCheck components?

A. Inspection Procedure

Our rules generated over 24K violations. It is not feasible to manually inspect all these violations, so we used the following sampling procedure. For violations from AutoEP (total of 24,472), we sampled two violations for each AutoEP mode, i.e., we inspected 84 violations. For violations from DefectsEP (total of 348), we grouped the violations based on which rules are violated, which tools violate the rules, and on which projects the rules are violated. In total, we create 25 groups, from which we inspected 41 violations. We provide in-depth description and the exact groupings of violations on this paper's companion website [54]. Finally, for the violations observed by running evolving programs obtained by EvoEP we inspected all 82 violations. While not all violations inspected may indicate bugs in RTS tools, our inspection found only two false positives among the violations.

B. Detected Bugs

In total, we discovered 27 real bugs from our inspection, with only two false positives. Table VII shows the list of bugs detected by RTSCheck. We group the bugs into one of three types based on inspected violations. Each row in the table describes one bug. Column 1 shows a unique bug Id. Column 2 is a short description of the bug. Column 3 is the type of the bug, which can either be an implementation bug (I) or a technique limitation (T). Column 4 shows the rule that was violated; Column 5 shows which components' evolving program triggered the bug. Finally, Column 6 shows the status of the bug: (1) "Confirmed" indicates that we reported a bug and developers confirmed our findings; (2) "New" indicates that we reported a bug, and developers did not yet respond; and (3) "Known" indicates that we found a bug that has been reported previously or known to developers.

C. ***RQ1***: *Safety Violations*

Answer: We discovered nine bugs in three tools and no false positive. AutoEP and EvoEP led to the discovery of eight bugs and one bug, respectively.

TABLE VII: Detected Bugs (I=Implementation Bug, T=Technique Limitation)

	Id	Description	Type	Rule	Component	Status
Safety	Clover-1	Moving a class does not update dependency cache	I	R1	AutoEP	Confirmed
	Clover-2	Accessing a field does not create dependency	T	R1	AutoEP	New
	Clover-3	Overriding a method not captured	T	R1	AutoEP	New
	Clover-4	Using a class with the instanceof operator not captured	T	R1	AutoEP	New
	Clover-5	Invoking a constructor and introspecting the class does not create dependency	T	R1	AutoEP	New
	Clover-7	Hiding a field not detected	T	R1	AutoEP	Confirmed
	Clover-8	Having an overloaded method and then changing a class hierarchy not detected	T	R1	AutoEP	Confirmed
	STARTS-1	Invoking tests via Suite does not create compile-time dependencies on tests	I	R1	AutoEP	Confirmed
	All-1	Does not detect changes to non-Java files	T	R1	EvoEP	Known
Precision	Clover-9	Invoking tests via Suite class not supported	I	R3	AutoEP	Confirmed
	Ekstazi-1	Invoking tests via a JUnit3 runner is not wrapped to capture dependencies	I	R5	DefectsEP	Known
	Ekstazi-2	Always selects two tests when run on Joda-time even if no changes between runs	I	R5	DefectsEP	New
	STARTS-2	Invoking all tests by creating a single Suite always runs all tests	I	R3	DefectsEP	Confirmed
Generality	Clover-6	Instrumenting classes under test changes program behavior	I	R7	AutoEP/DefectsEP	Known
	Clover-10	Cannot parse a subset of Java syntax	I	R6	DefectsEP	New
	Clover-11	Inserts incompatible code during instrumentation	I	R6	DefectsEP	New
	Clover-12	Instrumentation cannot deal with two classes that have same fully qualified name	I	R6	EvoEP	New
	Clover-13	Introduces external libraries that pollute shared cache	I	R7	EvoEP	New
	Clover-14	Parser does not properly support checker framework's annotations	I	R6	EvoEP	New
	Clover-15	Ignores tests explicitly requested to be executed in pom file	I	R6	EvoEP	New
	Clover-16	Not able to find a core Clover class at runtime due to problems with classpath	I	R6	EvoEP	New
	Ekstazi-3	Crashes due to incompatibility with outdated build systems	I	R6	DefectsEP	Confirmed
	Ekstazi-4	Unexpectedly triggers JUnit4 annotations under JUnit3 framework	I	R7	EvoEP	Confirmed
	Ekstazi-5	Improper support of @Inject annotations	I	R7	EvoEP	New
	STARTS-3	Incompatible with specific third-party libraries	I	R6	EvoEP	Confirmed
	STARTS-4	Cannot support tests in a non-conventional location on disk	I	R6	EvoEP	Confirmed
	STARTS-5	In-memory dependency graph grows out of available memory	I	R6	EvoEP	Known

We illustrate one bug discovered by AutoEP and one bug discovered by EvoEP; we simplify and format the code for ease of presentation.

1) AutoEP: **Field hiding (Clover-7)**. Figure 2a shows an evolving program that triggers a bug in Clover due to incorrect handling of field hiding. Clover misses to detect that a new field was added to the class of the instance used during test execution, thus skipping to select a failing test in the second revision. This bug was detected by inspecting a violation of rule R1 and the bug was confirmed by Clover developers.

2) EvoEP: **External dependencies (All-1)**. Figure 2b shows an evolving program that illustrates the limitation of RTS tools used in our study. In summary, EvoEP extracted an evolving program from the Graphhopper project. Several tests were accessing .txt files on disk, which is not captured by any RTS tool used in our study. This was detected by inspecting a violation of rule R1. Developers of RTS tools classified this case as a known bug and a limitation of the RTS techniques.

3) Discussion: We note that only evolving programs obtained by AutoEP and EvoEP discovered bugs due to safety violations. We expected that EvoEP would not detect many safety violations because public repositories rarely include failing tests. Within the violations inspected, no bug due to safety violation was discovered by DefectsEP. Our result shows the benefit of automated test generation and using only existing bug databases is not sufficient for checking RTS tools.

D. *RQ2: Precision Violations*

Answer: We discovered four bugs in the tools and identified two false positives. Two components (AutoEP and DefectsEP) led to the discovery of one and three bugs, respectively. The bugs were discovered in all RTS tools.

We describe one bug discovered by AutoEP, one bug discovered by DefectsEP, and a false positive reported by EvoEP.

1) AutoEP: **@Suite (Clover-9)**. Figure 2c shows an evolving program that led to a bug found in Clover. Clover always selects to run a test annotated with `@RunWith(Suite.class)`. The example violated rule R3. This is an implementation bug and was confirmed by Clover developers. Indeed, we found that any test generated by Randoop that includes `@Suite` would lead to this violation, resulting in an overwhelming number of violations. We eventually modified the default generation in Randoop to output tests without `@Suite`. Our total count of violations do not include those due to `@Suite`, but we note that we could find a bug in Clover due to Randoop generating such tests.

2) DefectsEP: **Lack of JUnit3 support (Ekstazi-1)**. Figure 2d shows an evolving program that led to a bug found

```
class CTest {
   @Test void test() {
      C c = new C();
      assertEquals(10, c.f); }}
public class A {
   public int f = 10;
}
public class C extends A {
+  public int f = 11;
}
```

(a) Clover-7 bug

```
class CTest {
   public void test() {
      assertEquals(1,
        new A().readStatusCodeFromFile());
   }
}
public class A {
   private String filePath = "PATH/TO/FILE" ;
   public int readStatusCodeFromFile() {
      int status = .../* Read the status
      code from the file at filePath*/
      return status;
   }
}
```

(b) All-1 bug

```
class CTest {
   @Test void test1() throws
        Throwable {
      C c = new C();
      int x = c.m1();
      org.junit.Assert.assertTrue(x
          == 0); }}
@RunWith(Suite.class)
@Suite.SuiteClasses({ CTest.class })
class RegressionTest {}
class C {
   public int m1() { return 0; }
}
```

(c) Clover-9 bug

```
import junit.framework.*;//JUnit3
public class CTest extends TestCase{
   public static Test suite() {
      return new TestSuite(CTest.
           class);
   }
   public void test() {
      assertNotNull(new A());
   }
}
```

(d) Ekstazi-1 bug

```
class CTest {
   public void test() {
      assertEquals(1,
        B.class
        .getDeclaredClasses()
        .length);
   }
}
public class A {}
public class B {
   public A a;
   public void m() {}
}
```

(e) Clover-6 bug

```
...
  <dependency>
    <groupId>
      com.h2database
    </groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
    <scope>test</scope>
  </dependency>
...
```

(f) STARTS-3 bug

Fig. 2: Several examples of evolving programs that illustrate bugs in RTS tools under test; each subcaption corresponds to a bug id in Table VII. The lines that are added are prefixed with "+"

in Ekstazi. This evolving program has tests in JUnit3 style, and if the test is run with Ekstazi twice, the test is executed both times. We noticed that this bug was fixed very recently in the latest release of Ekstazi, so it is a known bug. This bug was discovered because rule R5 was violated in the Apache Commons-lang project.

3) EvoEP: We found no bug due to precision violation from the evolving programs obtained by EvoEP. We illustrate and analyze the reason for a false positive.

RetestAll run after every n revisions. We discovered that Clover forces the execution of all tests every 10 revisions. Because this behavior is purposely implemented in Clover, we do not consider this violation to indicate a real bug.

4) Discussion: Evolving programs obtained by EvoEP did not discover any bug due to precision violations. At the same time AutoEP and DefectsEP discovered several bugs. These two components did not find any common bug, showing that both components are valuable and orthogonal.

E. ***RQ3****: Generality Violations*

Answer: We discovered 14 bugs in the tools and no false positive. Each component led to the discovery of at least one bug, one bug is discovered by more than one component, and the bugs were discovered in all RTS tools.

We describe one bug discovered by each component and compare the results of various components.

1) AutoEP: **Heavy instrumentation (Clover-6)**. Figure 2e illustrates an evolving program that led to a bug in Clover. Clover performs an intrusive instrumentation by inserting extra methods and fields in most of the classes. Any test that depends on the number of fields (e.g., via reflection) fails as there are now more fields than expected. This bug is discovered with an evolving program obtained by AutoEP, violating rule R7 when using state comparison as a test oracle. To avoid excessive number of violations due to the same reason, we excluded fields added by Clover from subsequent state comparisons after we discovered this bug.

2) DefectsEP: **Heavy instrumentation (Clover-6)**. DefectsEP led to the discovery of the same bug as described for AutoEP. The bug was found because an evolving program extracted from the Joda-time project violated rule R7. This is the only bug that was discovered by more than one component in the inspected set of violations.

3) EvoEP: **Incompatible with a third-party library (STARTS-3)**. Figure 2f shows the build configuration script that exposed a bug in STARTS. Any program with this configuration leads to a `NullPointerException` in STARTS. This bug was discovered because rule R6 was violated in the DBCP project. We categorize it as an implementation bug, which is confirmed by the developers of STARTS.

4) Discussion: Our results show that all the components were able to detect generality violations. More importantly, based on the discovered bugs, we found that only one bug was reported by more than one component. This emphasizes the value of each individual component in our framework.

F. ***RQ4****: Efficiency Report*

Answer: Only DefectsEP and EvoEP are useful for checking efficiency of RTS tools; we find that Clover is inefficient and frequently takes a longer time to run than RetestAll.

Evolving programs generated by AutoEP have tests that take negligible time. Therefore, we did not find it appropriate to use AutoEP to check and compare efficiency of RTS tools.

Figure 3 shows efficiency reports for running DefectsEP and EvoEP. The figures show for each project the cumulative time (for all revisions of all evolving programs) taken by RetestAll,

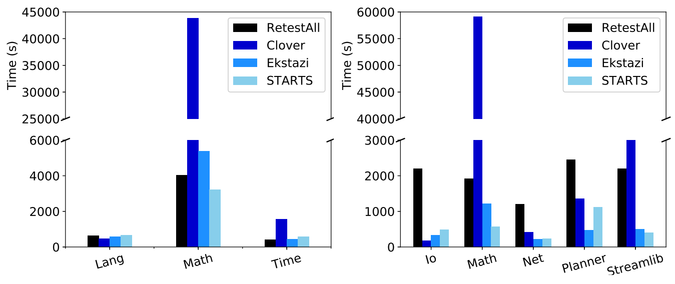

Fig. 3: DefectsEP and EvoEP efficiency report

Clover, Ekstazi, and STARTS. When computing cumulative time, we excluded those projects on which at least one RTS tool crashes on at least one revision. This way, if an RTS tool crashes early (e.g., due to a generality violation), we do not report the other tools as being relatively slower.

Based on the figures, we see that over all projects Clover is, in general, rather inefficient in terms of testing time. Clover takes more time than RetestAll on three out of seven projects. Additionally, we find that, on average, Ekstazi and STARTS both outperform RetestAll, but it is inconclusive which tool is more efficient based on the set of evolving programs used in our study. Future work could explore how to combine the benefits of these tools and how to automatically predict what tool would perform the best for a given context.

V. Discussion and Future Work

Extending RTSCheck. Once the framework was in place, it was relatively easy to include new rules and subjects. During our development of RTSCheck, when we needed to add more rules, we estimate it took us about an hour to add a new rule that uses the data and logs collected already. Adding new Maven projects to evaluate is trivial.

Mutation testing. An alternative approach to checking RTS tools is to apply mutation testing on large existing projects [46], [48], [58]. AutoEP component's evolution operators are similar to mutation testing operators, which may also be used to simulate code changes for use in RTSCheck. Unfortunately, we found that mutation testing is currently not feasible. The ideal mutation tool for checking RTS tools should *not* use mutant schemata [59] (because code must "evolve"), should mutate source code (some RTS tools analyze sources), work with various Java versions, and perform mutation statically. To the best of our knowledge, no existing mutation tool satisfies all these requirements.

Nevertheless, we performed an initial experiment on two open-source projects: Apache CSV and Google compile-testing. We mutated bytecode of these projects using PIT [49], resulting in 585 and 961 mutants, respectively. We *semi-automatically* translated these mutants to source code to obtain evolving programs. We then ran all three RTS tools on the evolved programs. None of the bugs detected by RTSCheck were detected by mutation testing, providing an initial evidence that mutants generated by traditional mutation testing do not help expose bugs in RTS tools.

Effective generated tests. We investigated what test length is the most effective in finding safety and generality violations with AutoEP; recall (Section II-B2) that we used four values for the maximum test lengths: {1, 2, 4, 100}. We had two findings. First, tests generated when the maximum sequence length was set to 4 led to the largest number of failing tests. Shorter tests are likely to depend on a small number of code elements, and thus more likely to expose safety and generality violations. Also, too short sequences (length = 1) may not execute enough code to lead to interesting dependencies. Second, we found a few cases when a violation was only revealed with very short tests (e.g., JD3+E6).

Flaky tests. Due to the limited Java model (e.g., no static fields) used for program generation, AutoEP does not generate flaky tests [22], [60], [61]. As described earlier, we filtered out flaky tests for DefectsEP. Finally, we did not observe flaky tests in our runs of EvoEP.

Execution cost. We executed AutoEP experiments in parallel on a supercomputer (using up to 256 nodes); each node has Intel Xeon Phi 7250. Program generation (excluding time in the queue), computed as if it was run sequentially, took a bit over five CPU days. Evolving those programs took over two CPU days. Finally, executing 31,104 evolving programs with RetestAll and three RTS tools took 139.3 CPU days. Experiments with DefectsEP took 19.2 CPU hours and EvoEP took 32 CPU hours. DefectsEP and EvoEP experiments were run on a 4-core Intel i7-6700 CPU @ 3.40GHz machine with 16GB of RAM, running Ubuntu 17.04.

Future work. RTSCheck can be improved and our infrastructure used as a basis for testing various *incremental* program analysis techniques. We plan to explore other ways of program generation, test generation, and evolution, as well as clustering evolving programs that expose the same bug. We plan to develop better strategies for grouping violations for inspection such that violations in the same group indicate the same bug(s). We plan on developing new rules and improving on our existing ones. As described in Section II-A, our rules are designed to be extreme to favor reducing false positives at the risk of missing true positives. We plan on investigating better thresholds for differences in tests selected as to better balance trade-offs between true and false positives. For existing rules, we plan to also expand them by inspecting which individual failed tests are missed to be selected by an RTS tool, not just the number of failed tests. Furthermore, we plan to look deeper at each failed test to see if the test fails for the same way as if run in RetestAll, e.g., fails for the same assertion or, in the case of AutoEP, the state captured at the second revision matches what was captured for RetestAll.

VI. Threats to Validity

External. Our framework may not readily generalize to RTS tools developed for other programming languages, e.g., C# [62]. Our current implementation supports only Java, but our methodology can apply to any programming language. We limited our experiment with DefectsEP to only the Defects4J bug database. We plan to integrate other bug databases in the future, e.g., Bugs.jar [63]. Projects that we used for EvoEP may not be representative of all projects. To mitigate this threat we chose popular open-source projects from GitHub that use Maven and were used in recent work on regression testing. We applied our framework to three RTS tools. We used the default configuration of RTS tools, as well as for Randoop and JDolly. Our reasoning is that the developers or tools that we used tuned default configurations to obtain optimal performance. We limited the number of generated tests to 50 per program for AutoEP. The limits on the number of programs and tests were set to make the experiments feasible. Although we used only one evolving program per project with EvoEP, we configured EvoEP to extract evolving programs from the most recent revisions of used projects.

Internal. RTSCHECK implementation or any scripts we wrote to run experiments may contain bugs. To mitigate this threat, we reviewed code and wrote unit tests.

Construct. We define seven rules and inspected violations of these rules to find bugs in RTS tools. We sampled the violations based on our own experiences with developing RTS tools. Furthermore, we found very few false positives in terms of violations indicating real bugs. Several of the bugs we found were confirmed by developers as real bugs.

VII. RELATED WORK

Regression test selection. RTSCHECK already finds bugs in dynamic and static approaches. Other techniques exist which compute test dependencies and affected tests in different ways. For example, AutoRTS [11] is a static, compiler-based technique which computes the dependencies of a test, T, from all the classes that must be compiled before T can be compiled.

RTS tools compute dependencies and affected tests using analysis at different granularity levels. Whereas STARTS and Ekstazi find test dependencies and affected tests at the class level, Clover works at the method level. Future work should include results on (1) more method-level RTS tools (e.g., Chianti [43], FaultTracer [44]), (2) hybrid class-and-method level RTS techniques like HyRTS [13], (3) hybrid class-and-statement level RTS techniques like DejaVOO [17], (4) statement-level techniques like Pythia [64], (5) module-level techniques like GIB [65], and (6) tools that capture dependencies across JVM boundaries like RTSLinux [66].

Automated test (input) generation. Bounded exhaustive techniques generate all test inputs up to a specified bound [67], [68]. TestEra [69] and Korat [70] generate test inputs based on imperative predicates. ASTGen [37] was the first approach for automatically testing refactoring engines; it used a framework for iterative generation of structurally complex tests. UDITA [71] introduced an expressive specification language to enable combining imperative predicates and iterative generation. Csmith [36] is a randomized compiler-testing tool for C. JDolly is the most recent work on testing refactoring engines [27]. RTSCHECK systematically uses JDolly to generate the first revision of each evolving program, but a future direction is to utilize other existing tools for program generation. We applied JDolly to a new domain, and we introduced new evolution operators and test oracles.

Pacheco et al. [30] presented Randoop. Search-based techniques are another popular approach for generating sequences of method calls [58], [72], [73]. We chose Randoop due to our familiarity with the tool.

Testing software engineering tools. RTSCHECK is the first approach for checking RTS tools. Similar to prior work in checking correctness of software engineering tools, we also rely on program and test generation. Mongiovi et al. [74] combined JDolly and Randoop to detect non-behavior preserving refactoring transformations. Cuoq et al. [75] used Csmith to test Frama-C. Kapus and Cadar [76] used Csmith to test symbolic execution engines. Recently, Dutta et al. [77] used a template based approach to generate programs and data for testing probabilistic programming systems. We differ from all these prior work in that we need to generate evolving programs (not a single program version).

Program transformations. Our proposed evolution operators are similar to operators used for mutation testing. Mutation testing has been traditionally used to evaluate the quality of test suites [45], [46], [48], [78]. Many mutation testing tools have been developed over the years, including Javalanche [79], Major [47], muJava [80], and PIT [49]. While we have several operators that are similar to existing mutation testing operators, we define unique operators made for the purpose of exercising interesting parts of the language to evolve the program in ways as to stress RTS tools.

VIII. CONCLUSION

Recent interest from industry, evidenced by recent adoption of RTS tools, has created a need to check and compare RTS tools more properly and systematically. RTSCHECK feeds evolving programs (i.e., sequences of program revisions) to RTS tools and checks the output against rules that specify potential violations of expected behavior. We applied RTSCHECK on three RTS tools, obtained 31K evolving programs, and detected 24K violations of the rules. We inspected 207 violations, from which we discovered 20 implementation issues and seven limitations of the underlying RTS techniques. We reported all 27 bugs to the developers of these RTS tools, who already confirmed 14 of them, 10 of which were previously unknown. Each RTSCHECK component contributed to discovering several unique bugs, and we found at least six bugs in each RTS tool. For researchers, RTSCHECK provides a framework to check correctness of future RTS tools or variants of the existing tools. For developers, RTSCHECK can provide confidence in RTS tools they may want to adopt.

Acknowledgments. We thank the anonymous reviewers for their comments. We thank Shirley Liu, Darko Marinov, Aleksandar Milicevic, Pengyu Nie, and Gregg Rothermel for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1421503, CCF-1566363, CCF-1652517, CCF-1704790, CCF-1763788, CNS-1646305, and CNS-1740916. We gratefully acknowledge support for research on testing from Microsoft and Qualcomm.

References

[1] P. Gupta, M. Ivey, and J. Penix. (2011) Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[2] N. York. (2011) Tools for continuous integration at Google scale. http://www.youtube.com/watch?v=b52aXZ2yi08.

[3] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *International Conference on Software Engineering, Software Engineering in Practice*, 2016, pp. 11–20.

[4] 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*, 2014, pp. 235–245.

[5] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[6] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica (Slovenia)*, vol. 35, no. 3, pp. 289–321, 2011.

[7] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Product-Focused Software Process Improvement*, 2010, pp. 3–16.

[8] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Journal of Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[9] P. Lakshman. (2017) Accelerated continuous testing with test impact analysis. https://blogs.msdn.microsoft.com/devops/2017/03/02/accelerated-continuous-testing-with-test-impact-analysis-part-1.

[10] Atlassian. (2018) About test optimization. https://confluence.atlassian.com/clover/about-test-optimization-169119919.html.

[11] J. Öqvist, G. Hedin, and B. Magnusson, "Extraction-based regression test selection," in *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–10.

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

[13] L. Zhang, "Hybrid regression test selection," in *International Conference on Software Engineering*, 2018, pp. 199–209.

[14] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic Regression Test Selection," in *International Conference on Automated Software Engineering (Tool Demonstrations Track)*, 2017, pp. 949–954.

[15] "Apache Camel - Building," http://camel.apache.org/building.html.

[16] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection techniques," in *International Conference on Software Engineering*, 1994, pp. 201–210.

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

[18] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[19] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering Methodology*, pp. 173–210, 1997.

[20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *International Conference on Software Engineering*, 2018, pp. 433–444.

[21] J. Bell and G. E. Kaiser, "Unit test virtualization with VMVM," in *International Conference on Software Engineering*, 2014, pp. 550–561.

[22] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[23] J. Bell, G. E. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *Symposium on the Foundations of Software Engineering*, 2015, pp. 770–781.

[24] 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*, 2015, pp. 223–233.

[25] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.

[26] M. Parfianowicz. (2017) Open Clover. https://openclover.org.

[27] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2013.

[28] W. F. Opdyke and R. E. Johnson, "Refactoring: an aid in designing application frameworks and evolving object-oriented systems," in *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990, pp. 145–161.

[29] G. Soares. (2018) JDolly program generator. https://github.com/gustavoasoares/jdolly.

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

[31] Randoop. (2018) Randoop home page. https://randoop.github.io/randoop/.

[32] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *International Conference on Quality Software*, 2003, pp. 20–23.

[33] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[34] A. Balestrat. (2018) CCG - random C Code Generator. https://github.com/Mrktn/ccg.

[35] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

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

[38] G. Soares, "Making program refactoring safer," in *International Conference on Software Engineering*, 2010, pp. 521–522.

[39] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Conference on Programming Language Design and Implementation*, 2017, pp. 347–361.

[40] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 386–399.

[41] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *International Conference on Software Engineering*, 2016, pp. 203–213.

[42] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[43] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of Java programs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 432–448.

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

[45] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, pp. 279–290, 1977.

[46] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[47] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Automated Software Engineering*, 2011, pp. 612–615.

[48] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[49] H. Coles. (2018) PIT Mutation Testing. http://pitest.org/.

[50] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[51] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using PageRank," in *International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[52] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *International Conference on Software Engineering*, 2018, pp. 537–548.

[53] M. Hashimoto, A. Mori, and T. Izumida, "Automated patch extraction via syntax- and semantics-aware delta debugging on source code changes," in *International Symposium on Foundations of Software Engineering*, 2018, pp. 598–609.

[54] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. (2019) RTSCheck. http://cozy.ece.utexas.edu/rtscheck.

[55] M. Gligoric. (2017) Ekstazi. http://www.ekstazi.org.

[56] STARTSTeam. (2018) STARTS - A tool for STAtic Regression Test Selection. https://github.com/TestingResearchIllinois/starts.

[57] Oracle and/or its affiliates. (2018) jdeps. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html.

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

[59] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *International Symposium on Software Testing and Analysis*, 1993, pp. 139–148.

[60] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 1–23.

[61] A. Zaidman and F. Palomba, "Does refactoring of test smells induce fixing flaky tests?" in *International Conference on Software Maintenance and Evolution*, 2017, pp. 1–12.

[62] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .NET," in *Symposium on the Foundations of Software Engineering, industry track*, 2017, pp. 848–853.

[63] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world Java bugs," in *International Working Conference on Mining Software Repositories*, 2018, pp. 10–13.

[64] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *International Conference on Reliability, Quality and Safety of Software-Intensive Systems*, 1997, pp. 3–21.

[65] V. Kosar. (2019) gitflow-incremental-builder (GIB). https://github.com/vackosar/gitflow-incremental-builder.

[66] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across JVM boundaries," in *International Symposium on Foundations of Software Engineering*, 2017, pp. 809–820.

[67] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *International Symposium on Software Testing and Analysis*, 2004, pp. 133–142.

[68] D. Jackson, *Software abstractions: logic, language, and analysis*. MIT Press, 2006.

[69] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Automated Software Engineering*, 2001, pp. 22–31.

[70] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *International Symposium on Software Testing and Analysis*, 2002, pp. 123–133.

[71] 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*, 2010, pp. 225–234.

[72] P. Tonella, "Evolutionary testing of classes," in *International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[73] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[74] M. Mongiovi, "Scaling testing of refactoring engines," in *Companion Proceedings of International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 15–17.

[75] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *International Conference on NASA Formal Methods*, 2012, pp. 120–125.

[76] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *Automated Software Engineering*, 2017, pp. 590–600.

[77] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Symposium on the Foundations of Software Engineering*, 2018, pp. 574–586.

[78] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic, "Approximate transformations as mutation operators," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 285–296.

[79] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for Java," in *International Symposium on Foundations of Software Engineering*, 2009, pp. 297–298.

[80] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Journal of Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.