Build System Aware Multi-language Regression Test Selection in Continuous Integration

Daniel Elsner
Roland Wuersching
Markus Schnappinger
Alexander Pretschner
firstname.lastname@tum.de
Technical University of Munich
Munich, Germany

Maria Graber
René Dammer
Silke Reimer
{grm,rda,sre}@ivu.de
IVU Traffic Technologies
Berlin, Germany

ABSTRACT

At IVU Traffic Technologies, continuous integration (CI) pipelines build, analyze, and test the code for inadvertent effects before pull requests are merged. However, compiling the entire code base and executing all regression tests for each pull request is infeasible due to prohibitively long feedback times. Regression test selection (RTS) aims to reduce the testing effort. Yet, existing *safe* RTS techniques are not suitable, as they largely rely on language-specific program analysis. The IVU code base consists of more than 13 million lines of code in Java or C/C++ and contains thousands of non-code artifacts. Regression tests commonly operate across languages, using cross-language links, or read from non-code artifacts. In this paper, we describe our build system aware multi-language RTS approach, which selectively compiles and executes affected code modules and regression tests, respectively, for a pull request. We evaluate our RTS technique on 397 pull requests, covering roughly 2,700 commits. The results show that we are able to safely exclude up to 75% of tests on average (no undetected real failures slip into the target branches) and thereby save 72% of testing time, whereas end-to-end CI pipeline time is reduced by up to 63% on average.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Software testing, regression test selection, continuous integration

ACM Reference Format:
Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. Build System Aware Multi-language Regression Test Selection in Continuous Integration . In *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513078

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA*, https://doi.org/10.1145/3510457.3513078.

1 INTRODUCTION

Regression testing is regularly performed on software systems to ensure changes did not inadvertently affect existing system behavior [23]. The simplest, yet expensive strategy to perform regression testing, *retest-all*, is to execute every test case after each change. However, with increasingly large test suites and limited physical resources this approach is often too costly, especially in continuous integration (CI) testing [11, 13, 24, 37]. To reduce the costs of regression testing, regression test selection (RTS) [15, 22, 29, 31, 32, 34, 38] has been extensively studied since the 1970s [14].

RTS techniques are considered *safe*, if they select *all* tests that are affected by the changes to the code base, such as the changeset of a pull request. Therefore, they collect dependencies for each test. This is implemented either through static (e.g., class dependency graph) or dynamic (e.g., code instrumentation) program analysis [15, 16, 21, 22, 34, 38, 39]. These per-test dependencies are then used to determine the relevant test cases. However, the involved language-specific program analysis can be expensive, which is why existing safe RTS techniques often bear (prohibitively) extensive costs [11, 24, 30]. Lightweight, less intrusive yet *unsafe* RTS techniques use CI or version control system (VCS) metadata to select tests, but the underlying statistical models can only provide project-specific empirical safety trade-offs [7, 8, 11, 12, 20, 24, 30, 35].

IVU Traffic Technologies is one of the world's leading providers of public transport software solutions. The software system considered in this study accounts for approx. 13.5M Java and C/C++ lines of code (LOC). A large variety of domain specific language (DSL) source files and non-code artifacts including build and other configuration files, expected test results, and resources complement the code base. IVU maintains the ten most recent releases of the system on dedicated release branches. Before pull requests are merged into any of these release branches, they are thoroughly tested for regressions. However, compiling the entire code base and running all of the thousands of regression tests for each pull request can take up to several hours despite a high degree of parallelization within the CI pipelines. This results in intolerable feedback time and is economically infeasible: When the number of queued pull requests increases, developers often need to wait until the next day for test feedback. Consequently, reducing the overall testing efforts in pull requests through selective compilation and testing is required.

Probabilistic RTS techniques have already been successfully employed at IVU in CI pipelines for the main development branch [12]. However, applying these unsafe techniques to pull requests on

release branches bears fundamental risks: Support patches are directly built from these release branches every day, hence, code from those branches is potentially deployed into customer's infrastructure. Therefore, when testing pull requests for release branches, we aim for test selection that is as safe as reasonably achievable.

Yet, existing safe RTS techniques are not applicable in the given industrial context: The complexity of the software requires test scenarios at integration and system level, where tests commonly operate across languages and intensively use non-code resources. To the best of our knowledge, there is no RTS technique that addresses this problem for Windows environments, as opposed to Linux-only approaches [10]. Furthermore, prior RTS approaches select a test case if the checksum of any per-test dependency has changed. There are two shortcomings to this approach: First, these RTS techniques might miss tests in case of changes made to the build system. For instance, adding a new runtime dependency in a configuration file might completely change a test's behavior, even though checksums of previously recorded per-test code dependencies are unchanged. Second, these techniques assume that a fully compiled workspace is readily available. At IVU, compiling the Java code base already takes roughly half an hour on average. Building only the relevant modules that are affected by changes or contain selected tests can have a significant impact on end-to-end CI pipeline execution time—especially for small changesets.

In this paper, we propose a build system aware RTS technique which harnesses dynamic and static program analysis to collect file-level per-test dependencies across language boundaries: We combine language-agnostic system call tracing, Java class loader instrumentation, as well as static code and build dependency analysis. This yields more complete per-test dependencies than pure language-specific approaches, thus leading to safer module and test selection for compilation and test execution, respectively.

We evaluate our RTS technique on 397 pull requests and measure the time savings in the CI pipelines across five weeks, covering roughly 2,700 commits. In addition to traditional measures, such as the ratio of selected tests, we also consider the observed real-life end-to-end time saving, i.e., compilation plus test execution time. The results on two evaluation branches show our approach can select tests safely (no undetected failures slip into target release branches) and thereby saves on average 42% and 72% of test execution time, depending on how recent the release branch is. End-to-end CI pipeline time is further reduced by up to 63% on average, when compared to pull requests with full build and retest-all strategy. Although we evaluate our RTS technique in just one industrial context, we expect it to be applicable to other multi-language software systems. Due to the resulting shorter feedback cycles, our RTS approach is now deployed to all release branches at IVU.

2 CONTINUOUS INTEGRATION TESTING AT IVU TRAFFIC TECHNOLOGIES

This paper describes the optimization of the CI testing process for pull requests at IVU using a novel build system aware multi-language RTS approach. To better understand the context of this study, we first explain the system under test and the testing process for pull requests at IVU. Second, we elaborate on established state-of-practice RTS techniques and their drawbacks in the given context.

2.1 System Description

At IVU, source code of the main software products is stored in a monolithic code repository. The code repository is split into two subtrees, one containing mainly C/C++ source code (approx. 9.5M LOC) and one with mainly Java source code (approx. 4M LOC). Additionally, both subtrees contain, amongst others, hundreds of thousands LOC written in Java-/TypeScript, C#, Perl, Python, SQL, and Assembly, as well as millions of lines in non-code artifacts such as XML, CSV, YAML, Java Properties, and plain text files.

The code repository is structured through >4,000 Maven[1] modules, yet Maven is only used as the build system for the Java subtree. For the C/C++ subtree, a self-maintained build tool is employed, which wraps Microsoft's Visual C++ compiler, as most software products primarily target Microsoft Windows.

The regression test suite is composed of unit, integration, and system level test cases. Tests written in Java are naturally located in the Java subtree, whereas C/C++ tests reside in the C/C++ subtree. In this study, we focus on the Java tests. These are further separated into so-called *short-running* and *long-running* test cases. Both types of test cases frequently interact with databases, which makes them inherently costly to run. Opposed to short-running tests, long-running tests operate on real databases instead of in-memory databases and use the Java Native Interface (JNI) to interact with dynamic-link libraries (DLLs) built from the C/C++ subtree. Executing the entire suite (~10,000 test cases) yields unbearable feedback times of around 2 hours (excluding compilation and code analysis; see next section). Motivated by this high potential for improvement, this paper describes how we reduce the effort for building and testing Java code in pull request CI pipelines.

2.2 Pull Request CI Testing

To continuously integrate code changes, IVU uses a Jenkins[2] CI system. Respective CI pipelines (1) build, (2) analyze, and (3) test the code base. These pipelines are continuously running for the main development branch and release branches, which contain currently supported and already released versions of the software.

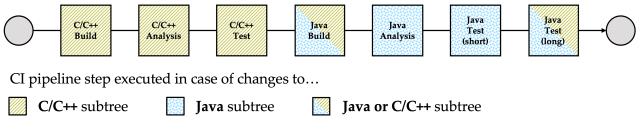

Figure 1: Pull request CI pipeline as executed after every change to a pull request branch

Before developers integrate changes into one of these branches, they have to create a *pull request*. These pull requests contain changesets which typically implement one feature, bug fix, or other enhancement. When a pull request is opened, a new CI pipeline is created for it. This pipeline will rebase the pull request branch (*source*) onto the *target* branch (e.g., the release branch) and execute

[1] Maven: https://maven.apache.org
[2] Jenkins: https://www.jenkins.io/

the above mentioned steps. Every new commit to the source branch triggers a new run of the pipeline. Since there exists only a limited amount of build machines for these pull request CI pipelines, a pipeline run is typically queued before being executed. Once a machine is available, the CI pipeline for the pull request runs as depicted in Fig. 1. First, in case there are any changes related to the C/C++ subtree, the C/C++ code is built, analyzed, and tested. Second, the Java code is built, analyzed, and tested. Notably, exclusively changing the C/C++ subtree nevertheless triggers the Java build and the Java long-running tests, as these tests operate across language boundaries and potentially use C/C++ DLLs[3]. If no changes to the C/C++ code were made, the necessary DLLs are loaded from a central artifact repository.

Before the improvements we describe in this study, the Java-related pipeline steps took around 170 minutes on highly parallelized build machines (64–96 cores): Building took on average 28 minutes, static code analysis 24 minutes, short-running tests 38 minutes, and long-running tests 80 minutes.

2.3 Alternative Test Selection Approaches and State-of-Practice

The state-of-practice knows several safe RTS techniques and associated tools, specifically for Java projects [2, 4, 10, 15, 16, 19, 21, 22, 27, 29, 33, 34, 38, 39]. Existing techniques statically [2, 21, 22, 33] or dynamically [4, 10, 15, 16, 27, 34, 38, 39] collect per-test dependencies at the level of basic-blocks [19, 29], methods [4, 39], classes/files [15, 16, 21, 22], modules [2, 33], or combinations thereof [34, 38] to select tests. These techniques have been shown to effectively reduce the testing effort in various studies—especially on open-source software projects, yet particularly not at the scale of the industrial code base of IVU. However, existing techniques and tools suffer the following limitations that are crucial in the context of IVU.

2.3.1 Build System Awareness. We identify two requirements related to the build system.

First, the majority of more recent RTS techniques use file or method checksums of per-test dependencies to identify those dependencies that have changed since the last test execution [10, 15, 16, 21, 22, 38]. While this approach is easily transferable and does not require integration with the VCS, it is based on the assumption that a fully built workspace is readily available. At IVU, compiling the code of all >2,000 Java Maven modules already requires a significant time effort, on average roughly half an hour despite high parallelization. However, selecting only relevant modules for compilation provides a great time-saving potential. Thus, an adequate RTS technique in this context needs to select tests without having compiled sources available.

Second, existing RTS techniques are unsafe considering the build system configuration. This is because they either use language-specific analysis or collect per-test dependencies during test execution. Yet, build configuration files such as Maven `pom.xml` files, are not part of per-test dependencies since these files define the compile and runtime dependencies *before* the test is executed. To mitigate this threat, Shi et al. [34] recommend to use hybrid techniques, which combine static module and dynamic file-level analysis. However, their proposed technique, `GIBstazi`, selects *all* tests, if *any* changes to non-code files occur. Due to the large amount of non-code artifacts in our system, this does not provide significant advantages over retest-all. At IVU, we require a more precise approach that actively checks for build system changes and prunes the set of selected tests and modules to be compiled to a minimum.

2.3.2 Multi-language Test Traces. Since most existing RTS techniques for Java software rely on language-specific static analysis or code instrumentation, their collected per-test dependencies are impracticable for software that operates across language boundaries [10, 40]. At IVU, we intensively use cross-language links to implement Java tests that cover code from the Java and C/C++ code base, and code written in other programming languages. Moreover, our code base includes millions of lines in non-code artifacts, which are used for configuration purposes at run-time or serve as test resources (e.g., expected test results). Consequently, a safe RTS technique in our context needs to rely on test traces that cover multiple languages and non-code artifacts, e.g., by tracing calls issued to native DLLs during test execution.

To address this problem, Celik et al. [10] propose `RTSLinux`, an RTS technique that modifies the Linux kernel to intercept operating system calls related to file accesses and process management. This way, `RTSLinux` collects all file dependencies of a Java virtual machine (JVM) process executing a test and is thereby also aware of calls made from the JVM to native libraries via the JNI. However, even if `RTSLinux` was publicly available, it would not be applicable in our case: First, perhaps trivially, our software targets Windows environments and therefore—at least the C/C++ parts—cannot be executed on Linux. Since Windows is a closed-source operating system, directly modifying the kernel and system call application programming interfaces (APIs) is not easily possible. Second, assuming said kernel modification was technically feasible, IVU would most probably decide against using modified operating systems on their CI machines, as this would imply maintaining that extension for future versions of the kernel with utmost care to avoid kernel panics. Third and yet more importantly, `RTSLinux` also relies on file checksums to compute the set of changed files and thus requires a compiled workspace. Hence, `RTSLinux` does neither provide an applicable approach for multi-language software on Windows, nor does it address the need to reduce compile time.

2.3.3 Tool Support. Beyond the conceptual limitations of existing RTS techniques, we did not find a single RTS tool for JVM projects that was publicly available (released on Maven Central) and worked out-of-the-box in the given context. The main reasons are that existing tools do not support JUnit 5 (`Ekstazi` [15], `HyRTS` [38]), or fail with Java Development Kit (JDK) versions newer than 9 (we use 11) or specific language features such as Java type annotations[4] (`Clover` [4], `STARTS` [22]).

In summary, we need an RTS solution that is (1) build system aware, i.e., it is safe concerning changes to the build system configuration and can selectively build only those modules relevant for testing

[3] Although C/C++-only changesets are supposed to trigger the Java build and Java long-running tests since they might break tests, this has been partially deactivated throughout the course of our study, due to the significant overhead caused for small C/C++ changesets.

[4] Clover GitHub Issue on Java Type Annotations: https://github.com/openclover/clover/issues/20

introduced changes; (2) capable of collecting per-test dependencies to non-code artifacts and across programming languages.

Similar to prior RTS research [12, 15, 16, 21, 22], we target class- rather than method-level test selection. Therefore, unless otherwise stated, we refer to a Java test class (i.e., JUnit test suite) in the rest of the paper when we talk about a *test*.

3 BUILD SYSTEM AWARE MULTI-LANGUAGE TEST SELECTION

Existing RTS techniques are either unsafe or unapplicable at IVU. This motivates a novel approach, which is build system aware and capable of multi-language RTS. In this section, we first explain how to collect multi-language test traces as input for the test selection. Second, we show how to integrate the test selection with the build system to selectively compile modules for safe and cost-efficient testing. Last, we elaborate on integration details of our RTS technique into the pull request CI at IVU.

3.1 Collecting Multi-language Test Traces

To address the requirement of multi-language support for our RTS technique, we need to collect per-test dependencies to non-code artifacts and across language boundaries. Therefore, we harness and combine practical approaches for system call tracing and JVM class loader monitoring. By integrating these approaches with the JUnit testing framework [5] and the Maven Surefire Plugin [3], we can collect the required test traces at file-level granularity.

3.1.1 Probe-based System Call Tracing. System calls represent the interface to the operating system kernel that is visible to application programmers [36]. During a test's execution, its low-level behavior can be analyzed by tracing the invoked system calls. Thereby, we can collect the set of all accessed files, even if they are accessed by loaded DLLs or transitive child processes. The most straightforward way to trace all system calls issued by a test is to execute each test in isolation in its own forked JVM process which is supported by standard test execution frameworks [3, 10]. This further increases the reliability of test results as it prevents shared test state pollution or test-order dependencies [6, 28]. We can thus obtain stable file-level test traces by tracing all process- and file-related system calls for each JVM, i.e., for each test.

A similar approach is employed by `RTSLinux` [10]. However, none of the techniques for tracing system calls evaluated by Celik et al. [10] is available for the Windows operating system. As motivated in Sec. 2.3.2, even if `RTSLinux` was available for Windows, we would prefer a less intrusive, more maintainable approach. Yet, from the results reported by Celik et al., we learn that an efficient system call tracing approach has to operate in *kernel mode*, since tracing *all* system calls and filtering in *user mode* is prohibitively expensive (approx. four times the overhead of kernel mode tracing) [10].

In order to implement practical and efficient system call tracing, we use `DTrace` [9][5]. `DTrace` provides capabilities to dynamically instrument so-called *probes*. These are static or dynamic instrumentation points for which one can specify instructions to be executed if the probe fires (e.g., when entering a system call). Because this selective probe-based instrumentation is highly efficient and guaranteed to run safely inside the kernel [18], it has been deployed in production environments at Netflix, amongst others [17].

Our `DTrace` script takes a process identifier (PID) as input and instruments relevant system calls related to accessing files or spawning new processes. This way, we capture complete traces and finally store all relevant information (e.g., timestamps, PIDs, accessed filepaths) in a *tracing log*.

3.1.2 JVM Class Loader Monitoring. There are two drawbacks to relying *solely* on tracing system calls for Java tests:

First, in case a test loads a Java `.class` file that is located inside a Java archive (JAR), that JAR file will be part of the test trace, but not the actually used `.class` file. This could lead to imprecision in the test selection, as it stipulates to select that test if any of the files inside that JAR has changed [15]. Since many of our tests use classes from other Maven modules which are typically packaged as JARs, addressing this potential imprecision is crucial. Therefore, we attach a Java agent[6] to each JVM executing a test. The agent monitors whenever a new class is loaded by the class loader. If the corresponding `.class` file is located inside a JAR, it is added to the tracing log. If it is not located inside a JAR, we can safely ignore it, as we already cover it with our system call instrumentation. Note that we are only interested if a `.class` file was *ever* loaded during the execution of a test. Hence, as every test is running in its own forked JVM, we do not need to instrument the loaded file itself.

Second, similarly, in case a resource, such as an XML file, is loaded that is located inside a JAR, no separate system call to open that resource is invoked by the JVM. Instead, it is read from the already loaded JAR. Therefore, we instrument the `getResource(String)` method in the `java.lang.ClassLoader` class, as it is used for loading resources from JARs.

3.1.3 Integration with Testing Infrastructure. At IVU, we rely on JUnit 5 as our testing framework for Java. To execute each test in isolation in its own JVM process, we use Maven Surefire's forking mechanism[7]. This creates a new JVM per JUnit test class and we parallelize testing across Maven modules on all available CPU cores. To link the individual tests to the accessed files and spawned processes from our tracing log, we further need to obtain information about when a test started and terminated. Therefore, we register a custom JUnit test execution listener and subscribe to a test's start and end event [5]. The listener creates a *testing log* which contains start and end timestamps, the identifier of the test, as well as the JVM PID. We cannot only use the PID of the test to find its file accesses, as Windows may reuse PIDs after a process has been terminated.

Eventually, the tracing and testing logs are combined and we are able to compute a file-level test trace for each test[8] after all tests have been executed. We store the test traces in a CSV file, where each row contains a test name and a filepath accessed by that test.

[5] `DTrace` stands for Oracle Solaris Dynamic Tracing Facility

[6] Java Agent API for run-time code instrumentation on the JVM [1]

[7] `mvn surefire:test -DforkCount=1 -DreuseForks=false -T1.0C`

[8] Recall that we refer to a Java test class (i.e., JUnit test suite) when we talk about a *test*

3.2 Build System Aware Test and Module Selection

Our proposed RTS technique acknowledges changes to non-code artifacts and to the build system, e.g., changed dependencies. Algorithm 1 contains the pseudo-code of our build system aware test and module selection. Our algorithm has three inputs: (1) the changeset from the VCS, (2) the multi-language file-level test traces described in Sec. 3.1, and (3) *DLL-to-source* mappings for target and pull request branch. The latter are retrieved by analyzing the C/C++ compiler output which contains static compile dependencies for each DLL. More specifically, we parse `.tlog` files that are emitted by the Microsoft C++ compiler toolchain[9] and extract which source files each DLL depends on. The resulting mapping is stored in a CSV file, where one row has two columns; one contains the DLL file name and one a source filepath this DLL depends on. We need the information contained in this mapping since our test traces only include accesses to DLL files, not to actual C/C++ source files.

Changeset Analysis. Our algorithm analyzes the changeset by iterating over all changed files. For each Java source file, we search the Java source file for all `class`, `enum`, and `interface` definitions. From those definitions we create corresponding `.class` filepaths that match `.class` files generated by the Java compiler. For instance, if a Java source file that is part of the package `a.b.c` contains two classes `X` and `Y`, the two generated class filepaths are `a/b/c/X` and `a/b/c/Y`. These will also match accessed `.class` filepaths of nested or anonymous classes, e.g., if `X` contained one anonymous class, the Java compiler would output a file `a/b/c/X$1.class`, which can also be matched by `a/b/c/X`. Then, we check for presence of JUnit test method annotations. If the file does indeed contain test methods, it is considered a test suite and thus added to the set of tests to be executed. This way we safely select all newly created or updated test classes. For each C/C++ source file, we retrieve all affected DLLs from the DLL-to-source mappings and add the DLL paths to the set of affected filepaths. For changed Maven `pom.xml` files or files that can affect the build results (e.g., `.wsdl` or `.xsd` files), we select all tests from the changed module and all downstream modules, by retrieving them from the Maven reactor. The filepaths of all other changed files (e.g., `.xml` or `.csv` files) are also added to the set of affected filepaths.

While our approach intuitively works for additive and modifying changes, it is not immediately clear, how deletions have to be treated for each file type: We address deletions of and inside of `.java` files by parsing both, the old and the new revision (if existing) of the file. This way, we can also find all affected tests that covered any `.class` filepaths of the old revision. Deletions related to C/C++ are already covered, since we use the DLL-to-source mappings from both, the target branch and the pull request branch. Hence, if the pull request contains a C/C++ deletion, the deleted filepath will still be part of the DLL-to-source mapping of the target branch. For deletions of other file types, we simply search for the old filepath inside the test traces.

Test Selection. We iterate over all *test traces* and select those tests that have accessed files that match any of the affected filepaths.

[9] `.tlog` files in Microsoft Visual C++ [26]

Algorithm 1: Build System Aware Test and Module Selection

Input: changeset, test traces, DLL-to-source mappings
Output: selected tests, selected modules

```
1  selectedTests ← {}
2  selectedModules ← {}
3  affectedFilePaths ← {}
4  foreach change in changeset do
5      if isJavaFile(change) then
           /* get .class filepaths from .java file (before/after) */
6          affectedFilePaths +← getClassFilePaths(change)
           /* if @Test annotation is present, select test suite   */
7          if containsTests(change) then
8              selectedTests +← getTestSuiteIdentifier(change)
9      else if isCppFile(change) then
           /* look up affected DLLs in DLL-to-source mappings     */
10         affectedFilePaths +← getDLLFilePaths(change)
11     else if isRelevantForBuild(change) then
           /* select all tests of changed Maven module and from all
              transitive downstream modules                       */
12         selectedTests +← getTestsForModule(change)
13     else
14         affectedFilePaths +← change
   /* select modules for changes in pom.xml, .java, .xsd, and
      .wsdl files                                                 */
15 if isRelevantForBuild(change) then
       /* get enclosing Maven module for compilation              */
16     parentModule ← getParentMavenModule(change)
17     selectedModules +← parentModule
       /* include upstream modules (transitive)                   */
18     selectedModules +←
         getUpstreamModules(parentModule)
       /* include direct downstream modules with their
          transtive upstream modules                              */
19     selectedModules +←
         getDownstreamModules(parentModule)
   /* compute affected tests from test traces                     */
20 selectedTests +← getAffectedTests(affectedFilePaths)
   /* search affected parent Maven modules for selected tests     */
21 testModules ← getAffectedModules(selectedTests)
22 selectedModules +← getUpstreamModules(testModules)
23 return selectedTests, selectedModules
```

Module Selection. We select all Maven modules for compilation that either (1) are directly impacted by the changeset or (2) enclose any of the selected tests. For (1) we further need to add all upstream modules to the set of selected modules, as they are required to compile the changed modules. Additionally, since changes in modules from (1) could break direct downstream modules, these also need to be selected including their transitive upstream modules, in order to be buildable. For (2) we need to add upstream modules as well, since they are required to build the tests' modules.

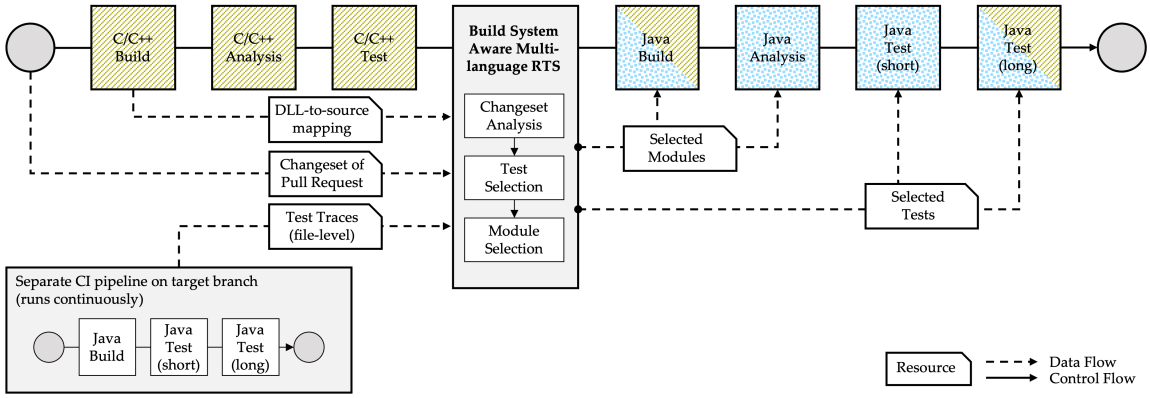

Figure 2: Pull request CI pipeline extended by our RTS technique

In the next section, we describe how we use these two results—the sets of selected tests and modules—inside the pull request CI to instruct Maven and Maven Surefire to selectively compile and execute modules and tests, respectively.

3.3 Integration into Pull Request CI

We integrated our RTS technique into the pull request CI, as illustrated in Fig. 2. Therefore, we added an additional step to the pipeline that calls our RTS algorithm, which we implemented as a simple command line interface (CLI) tool.

To provide the test traces to our tool, we created a separate *tracing* CI pipeline, that continuously runs for the target branches chosen in this study (see Sec. 4.1 for our evaluation setup). Currently, we update the test traces approx. once per day for each release branch, which provides good trade-offs regarding effort and effectiveness in our context (see our results in Sec. 4.2). We only update test traces for passing tests, since failing tests might yield incomplete traces (e.g., if the test terminated early).

Our tool receives the following inputs: First, the changeset of that pull request; second, the most recent test traces collected with our separate tracing CI pipeline; third, the DLL-to-source mapping from the most recent C/C++ build of the target branch. If changes were made to the C/C++ subtree within the pull request, we also provide the DLL-to-source mapping from the C/C++ build step. With these inputs, our tool computes the set of selected tests and modules and stores them in text files for subsequent pipeline steps.

To build and analyze only the selected Java modules, we extend the Maven reactor mechanism [25] to read the modules from that text file. We need this extension as the reactor API currently does not offer an option to specify the list of modules as a file[10]. To execute only the selected short- and long-running tests, we make use of Maven Surefire's test inclusion mechanism.

4 EVALUATION

The effectiveness of safe RTS techniques is typically evaluated by comparing the number of selected tests to the retest-all strategy and by measuring the overall test duration of the selected tests [10, 16, 21]. Regarding the evaluation of a technique's safety, prior work often semi-formally proves safety under certain assumptions [10, 16, 21], such as safety for code changes [16]. However, prior research on checking RTS tools has shown that these assumptions cannot be guaranteed to hold in practice [40]. We have already explained in Sec. 2.3 why non-code artifacts, cross-language links, and build system changes pose particular threats to the safety of existing RTS techniques. Thus, we need to empirically determine how safe our proposed RTS approach is for changesets of pull requests and discuss scenarios where our approach can be unsafe (see Sec. 4.3.2). Since no existing RTS technique is considered universally safe and could therefore serve as a reference, the only way to empirically check for safety violations is to find real (non-flaky) failures that were not selected by the RTS technique [34]. We perform an empirical study on five weeks of real development activity to answer the following research questions (RQs):

- **RQ$_1$**: How much testing effort reduction can we achieve by selecting tests using our RTS technique?
- **RQ$_2$**: How safe is our RTS technique for changesets of pull requests in terms of real missed failures?
- **RQ$_3$**: How much end-to-end CI pipeline execution time does our RTS technique save per pull request CI pipeline run?

4.1 Experimental Setup

In order to be able to answer RQ$_{1-3}$, we need to measure (1) testing effort reduction, (2) missed failures, and (3) end-to-end CI pipeline execution time savings for pull requests. We therefore added an invocation of our RTS CLI tool before the Java pipeline steps on the previous two release branches, R_{V_1} and R_{V_2}, and in the current release branch, R_{V_3}. Additionally, for all three branches we log the start and end timestamp for each Java pipeline step and the newly added RTS pipeline step. The build machines used for executing pull request pipelines are drawn from a fixed set of machines (64–96 cores), independent of the target branch of the pull request. Table 1 provides a summary of relevant descriptive statistics for the three release branches considered in this evaluation.

To answer RQ$_1$ and RQ$_2$ and measure (1) and (2), we only consider pull requests to R_{V_1} and R_{V_3}. On these branches, we still execute all tests and only store the set of selected tests for later

[10] Due to command line length limitations on Windows, we cannot use the `--projects` option.

Table 1: Relevant statistics for 3 considered release branches

Branch		R_{V_1}	R_{V_2}	R_{V_3}
Dataset Statistics	# PRs	88	58	251
	# PR pipeline runs	157	73	356
	# Commits	896	120	1,681
	Median changeset size (# files)	5	4	9

analyses (i.e., retest-all with *simulated* RTS). This way, we get a retest-all test report for each pull request CI pipeline run on these branches. From this report and the set of selected tests, we can compute the fraction of selected tests and test duration, and count the missed failures that occurred during retest-all. The rationale behind the choice of release branches is that at IVU release branches follow a specific lifecycle: On the current release branch, R_{V_3}, the most development activity is expected, as small features and bug fixes are still added. The less recent release branch, R_{V_1}, receives fewer development activity, which primarily concerns maintenance tasks, where pull requests with smaller changesets are expected. By using both R_{V_1} and R_{V_3} to answer RQ_1, we can also investigate to what extent RTS on pull requests to older release branches can achieve higher savings due to the smaller changesets.

To answer RQ_3 and measure (3), we need to compare end-to-end pipeline durations with RTS against retest-all. However, using both, RTS *and* retest-all on the same pull requests introduces large overhead—in the worst case a factor of 2, if all tests and modules are selected. In our industrial setting, this evaluation approach is rendered too expensive. From previous experience and analyses we know that CI pipeline runs take approx. the same time for pull requests to R_{V_2} and R_{V_3}, although they can be subject to natural variations due to infrastructure side-effects [12]. Hence, we can compare the time distribution for each pipeline step in R_{V_2}, where RTS is actually used, against R_{V_3}, where retest-all is used. In addition to this *inter-branch* evaluation approach, we validate our results by also using RTS on R_{V_3} for the final week of our experiments. This allows to investigate the pipeline runtimes before and after RTS activation in an *intra-branch* evaluation.

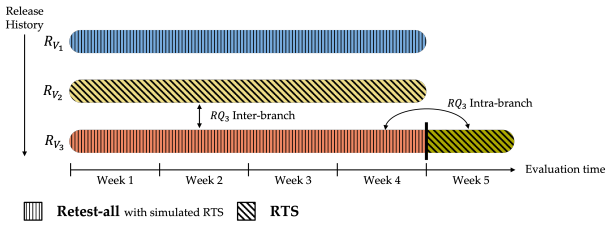

Figure 3: Evaluation approach for different release branches

The evaluation approach is illustrated in Fig. 3. Overall, our approach captures two aspects that have not been considered in prior research: First, we investigate the impact of the current lifecycle phase of a target release branch on RTS effectiveness. Second, we analyze how savings in each pipeline step contribute to the achieved end-to-end time savings for pull request CI pipeline runs.

4.2 Results

RQ_1: Testing Effort Reduction. Fig. 4 depicts the testing effort reduction we achieve, by comparing our RTS technique to a retest-all strategy. As described in the previous section, we compute the fraction of selected tests and test duration by combining retest-all test reports from pull requests on R_{V_1} and R_{V_3} with the respective sets of selected tests. The results indicate that our RTS technique selects on average 25% of tests for pull requests on R_{V_1} and 52% for R_{V_3}. The selected tests further take on average 28% and 58% of test duration for R_{V_1} and R_{V_3}, respectively. Hence, RTS was particularly effective for the maintenance branch R_{V_1}, which had smaller changesets (median changeset size is only roughly half of R_{V_3}).

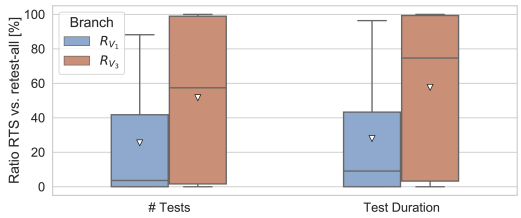

Figure 4: Comparison of our RTS technique to retest-all strategy regarding ratio of selected tests and test duration

RQ_1: *We find that on two evaluation branches our RTS technique selects on average 25% and 52% of tests per pull request CI pipeline run realizing a test duration reduction by 72% and 42%, respectively. RTS performs significantly better in pull requests on a maintenance release branch (R_{V_1}) compared to a release branch with active development (R_{V_3}).*

RQ_2: Safety. To find real failures which would not have been selected by our RTS technique, we first create the set difference of all failed tests in the retest-all test report and the selected tests. This yields a total of 305 pull request CI pipeline runs with missed failures across pull requests to R_{V_3} and R_{V_1}. To filter out failures that are *not* introduced by the changeset of the pull request itself, we need to re-run the failing tests at the revision of the target branch which the pull request was rebased on. If a test also fails on the target branch, it is probably not related to the pull request (e.g., a *won't fix*) and can be discarded. If a test does *not* fail on the target branch, we need to manually check if it is a failure introduced by the pull request and was therefore missed by the RTS technique. To keep the tedious manual effort at a reasonable level, we randomly sampled 50 from the 305 runs and manually inspected 2,176 missed test failures. Most of the missed failures stem from a database technology switch that was made on the build machines during the considered time period. This switch caused many long-running tests to fail due to memory leaks or failing schema updates during the first days of operation. We further observe a few flaky tests that failed due to non-deterministic test behavior that is partially known to the developers. However, none of the inspected missed test failures are actually related to the changes introduced in the pull requests. We can therefore conclude that our RTS technique

is empirically safe regarding failures that were introduced by the considered pull requests. Yet, we discuss potential reasons for safety violations in Sec. 4.3.2. We can further confirm previous findings that RTS techniques can be helpful in avoiding flaky test failures and thereby reduce associated debugging costs [34].

RQ$_2$: *We find that our RTS technique does not miss any real failures that are introduced in the considered pull requests.*

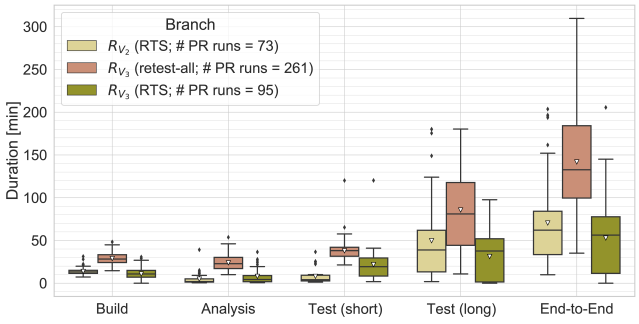

Figure 5: Inter- and intra-branch comparison of our RTS technique to full build and retest-all strategy

RQ$_3$: End-to-End Savings in Pull Requests. Fig. 5 shows the distributions of the duration for each Java pipeline step and the Java end-to-end runtime for R_{V_2} and R_{V_3}. As described in our experimental setup, we perform two kinds of comparisons, *inter-branch* and *intra-branch*. Regardless of the applied evaluation, we observe significant savings: In the inter-branch comparison, the results indicate that we can save on average 50% (71 minutes) of end-to-end pipeline runtime with RTS on R_{V_2}. In the intra-branch comparison, when comparing R_{V_3} (retest-all) against R_{V_3} (RTS), which was used in the final week of our experiments, we achieve even better results: On average, 63% (89 minutes) of end-to-end pipeline runtime is saved. Despite small discrepancies in the achieved time savings, our results show similar trends using either of the two evaluation techniques and thus confirm each other. Comparing the median time savings, there is only a discrepancy of 6 minutes between the evaluation methods.

Regarding the individual contributions to this overall end-to-end time, we report the following average savings for the individual pipeline steps for R_{V_2}: 53% for *Build*, 80% for *Analysis*, 79% for *Test (short)*, and 42% for *Test (long)*. We discuss the reasons for the comparatively smaller savings in long-running tests in Sec. 4.3.1.

We further find that computing the selected modules and tests is inexpensive: The mean and median of the RTS pipeline step was roughly 3 minutes across all three branches. Since we generate test traces and DLL-to-source mappings in *separate* CI pipelines every day, the end-to-end pipeline time for pull requests is unaffected.

RQ$_3$: *We find that our RTS technique helps save on average 50% and 63% end-to-end pipeline execution time for pull requests on two release branches.*

4.3 Discussion

In the following, we discuss weaknesses related to the precision and safety of our RTS approach and share feedback we received from developers working on the system.

4.3.1 Imprecision of DLL-to-source Mappings. Our results for RQ$_3$ indicate that time savings achieved for the long-running Java test step are lower than those for the short-running test step. The reason is that if there are changes in core modules of the C/C++ subtree, commonly the majority of long-running tests is selected. This implies that these changes affect any DLL used by many long-running tests. We do not have any runtime information about which C/C++ source files that are part of a DLL are actually covered by each test. Hence, our selection in such cases is rather coarse-grained and imprecise. To address this problem and obtain more fine-grained runtime information, we are currently investigating extensions to our approach such as instrumenting the DLLs, intercepting native function invocations from the JNI, or using `DTrace` for tracing additional relevant system events.

4.3.2 Potential Reasons for Safety Violations. Test traces and DLL-to-source mappings are created in separate CI pipelines continuously running on the target release branches. Depending on the frequency of these pipelines (we run them once per day), test traces and DLL-to-source mappings might be outdated, which in turn may lead to unsafe test selection. Similarly, if a test fails over multiple runs in the separate tracing CI pipeline, that test's trace will not get updated until it passes again, also leading to outdated test traces.

In case of changes related to dependency injection mechanisms, affected tests might be missed: For instance, if a new default Java EE bean implementation is added inside a pull request, all tests that use the default bean will change their behavior. Yet, none of the files inside the test trace is directly affected by the addition. However, in such cases, typically another change that affects the test trace is part of the pull request, such as adjusting any other file that uses the newly added bean implementation—hence, odds are low that we effectively miss any affected tests.

Eventually, our RTS technique might be unsafe, if new non-code artifacts are added to the code base that are implicitly used by tests, while tests are not changed themselves; for instance, if a test walks the file tree and opens all files with a certain file extension, rather than explicitly opening a file through its filepath.

Finally, we found that external configuration changes, e.g., to the database environment, can cause tests to fail. However, our RTS technique only considers artifacts tracked by the VCS and therefore did not select these tests. We believe this to be expected RTS behavior, since these failures are not related to the changeset of a pull request.

4.3.3 Developer Feedback. Since IVU engineers, architects, and testers are directly impacted by our changes to the pull request pipelines, we regularly asked them for feedback on our work. Overall, our RTS approach has wide support among developers as it significantly reduces feedback times in their daily work; they are convinced that the RTS approach adds great value. Therefore, we deploy it to all other release branches as well. Furthermore, as requested by developers, we are working on extending the test

selection to C/C++ tests. While the system call tracing is language-agnostic and the integration with the testing framework is straightforward, we require further language-specific instrumentation, as DLL test trace granularity is too coarse (see our discussion on imprecise DLL-to-source mappings above).

4.4 Threats to Validity

4.4.1 External Validity. As for most industrial case studies, the main threat to validity concerns the generality of our results: We have specifically designed our RTS approach to address the shortcomings of existing techniques in the context of IVU. Nonetheless, our results show similar trends as prior RTS research done on open-source software and we can confirm empirical findings that dynamic file-level RTS can indeed significantly reduce regression testing efforts [10, 16, 34]. Furthermore, the technology we use to implement our RTS tool is publicly available and we rely on standard frameworks and tools, such as JUnit and Maven, that are frequently used across research [10, 16, 21, 38]. This eases a replication of our study in other software projects.

Furthermore, similar to previous studies [12, 34], the measured times in the CI pipelines can contain irregular fluctuations stemming from infrastructure or environment issues. While this could affect our evaluation results, we address this threat by reporting not only (potentially biased) average values, but the distributions for time savings across analyzed pull request pipeline runs.

Finally, to assess the safety our RTS approach, we manually checked if there were any real missed failures, but limited the inspection to 50 randomly sampled pull requests, which might not be representative. However, as opposed to most prior studies on safe RTS, we discuss potential safety violations and perform an empirical study to find any occurrences. We do this even though we rely on concepts that have been shown to work in other safe RTS approaches. Furthermore, to the best of our knowledge, we have still re-run and inspected the largest number of missed test failures in any existing RTS study to date.

4.4.2 Internal Validity. The main internal threats emerge from the implementation of our RTS tool and the proper functioning of Maven Surefire, JUnit, `DTrace`, and the ByteBuddy library[11], which we use to instrument the Java class loader. To address these threats, we wrote unit and integration tests for our RTS tool and manually checked selection results of pull requests for their validity.

5 RELATED WORK

Throughout this paper, we have referenced RTS techniques that have been proposed to effectively reduce regression testing efforts (see Sec. 2.3). Among the many existing studies, we consider the following to be most relevant for our work:

Gligoric et al. [15, 16] propose `Ekstazi`, a dynamic file-level RTS technique for the JVM that relies on file checksums for computing the set of selected tests. In Sec. 2.3, we describe why `Ekstazi` is unsafe in our context, as it uses file checksums and is neither aware of cross-language links, nor does it consider non-code artifacts[12]. `Ekstazi` reduces the end-to-end testing time on average by 32% across 32 open-source projects. Furthermore, Gligoric et al. [16] find that selecting tests at the class level (i.e., JUnit test suites) achieves better results than selecting tests at method level (i.e., JUnit test methods). We acknowledge these results as our RTS technique also collects file dependencies per test class, rather than test method.

Shi et al. [34] extend `Ekstazi` by complementing it with the static incremental build tool `GIB` [2]. The resulting tool, `GIBstazi`, is thus the most similar existing approach to our hybrid approach of selecting modules and regression tests for compilation and test execution, respectively. However, `GIBstazi` selects *all* tests, if any changes to non-code files occur. Due to the large number of non-code artifacts in our system, this is too imprecise. Additionally, similar to `Ekstazi`, it is unsafe for changes to multi-language source or binary files, such as DLLs, and files accessed by those. Overall, `GIBstazi` achieves higher safety than `Ekstazi` and the empirical results on 22 open-source projects show that `GIBstazi` reduces end-to-end build and testing time in CI environments by 23%.

Celik et al. [10] propose `RTSLinux`, the first and only RTS technique that uses system call analysis to track accesses to files across JVM boundaries during testing. Similar to `Ekstazi`, `RTSLinux` uses file checksums for selecting tests, when a compiled workspace already exists. We have alluded to why `RTSLinux` is not applicable in Windows environments and why our lightweight, safe kernel instrumentation through `DTrace` is a more practical approach in an industrial setting than modifying the operating system kernel. `RTSLinux` saves 53% of test execution time compared to retest-all.

In a previous study at IVU, we evaluated the cost-effectiveness of unsafe RTS techniques that solely rely on readily available CI and VCS metadata [12]. When applied to the main development branch at IVU for six weeks, the best performing unsafe RTS technique achieved test time savings of on average 19.8% with 93.4% of failures being detected. Though, we have motivated before that for pull requests to release branches safe RTS is required.

In summary, we are not aware of any prior work that investigates safe RTS that is build system aware and operates across language boundaries. Moreover, neither do any of the previous studies evaluate safe RTS in a large-scale industrial CI setting, nor do they study how end-to-end time for pull request CI pipelines can be reduced.

6 CONCLUSION

At IVU, compiling, analyzing, and testing pull requests within CI pipelines has prohibitively long feedback times. To reduce testing effort for pull requests on release branches, *safe* RTS is required, since support patches for customers are directly built from release branches. However, existing safe RTS techniques are inapplicable, as tests at IVU commonly operate across languages and intensively make use of non-code artifacts. Moreover, prior RTS approaches are unsafe for changes to the build system configuration and require an already fully compiled workspace.

In this paper, we introduce a build system aware multi-language RTS technique that safely selects modules and tests for compilation and execution. We deploy our novel RTS technique in IVU's large-scale, multi-language code base and perform an extensive empirical study to evaluate its effectiveness. The results indicate that our RTS technique saves on average 42% and 72% of testing time on

[11] ByteBuddy: https://bytebuddy.net

[12] `Ekstazi` has a hidden Linux-only option to collect files loaded by the JVM, which is untested and disabled by default. Nonetheless, even when collecting files loaded by the JVM, file accesses made from native DLLs or transitive processes are missed [10].

two evaluation release branches. We thereby reduce end-to-end CI pipeline runtime for pull requests by up to 63% on average. Since this greatly reduces feedback time for developers while retaining failure detection, our introduced RTS technique is now deployed company-wide to all release branches. While our industrial case study provides insights for one specific context, we expect our RTS technique to be applicable to other multi-language software projects, as it is based on well-known concepts and widely used tools for dynamic and static program analysis.

ACKNOWLEDGMENTS

We thank Dennis Bracklow, Stefan Golas, Maximilian Pohl, and Stefan Sieber for their support while integrating our technique into IVU infrastructure. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant SOFIE 01IS18012B. The responsibility for this article lies with the authors.

REFERENCES

[1] 2017. Java Agent API. https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html
[2] 2021. gitflow-incremental-builder (GIB). https://github.com/gitflow-incremental-builder/gitflow-incremental-builder
[3] Apache Maven. 2021. Maven Surefire Plugin – surefire:test. https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html
[4] Atlassian. 2017. About test optimization. https://confluence.atlassian.com/clover/about-test-optimization-169119919.html
[5] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. 2021. JUnit 5 User Guide: Advanced Topics. https://junit.org/junit5/docs/current/user-guide/{#}launcher-api-listeners-custom
[6] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering*. 550–561. https://doi.org/10.1145/2568225.2568248
[7] Antonia Bertolino, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Breno Miranda, and Roberto Pietran-Tuono. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3377811.3380369
[8] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 975–980. https://doi.org/10.1145/2950290.2983954
[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. https://doi.org/10.5555/1247415.1247417
[10] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–820. https://doi.org/10.1145/3106237.3106297
[11] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910
[12] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834
[13] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. 1981. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*. 1–6.
[14] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.
[15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230
[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784
[17] Brendan Gregg. 2016. DTrace for Linux. http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html
[18] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.
[19] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. 2001. Regression test selection for Java software. *ACM SIGPLAN Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305
[20] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the International Workshop on Rapid Continuous Software Engineering*. 19–25. https://doi.org/10.1109/rcose.2015.11
[21] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361
[22] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710
[23] Hareton K.N. Leung and Lee White. 1989. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*. 60–69.
[24] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018
[25] Apache Maven. 2021. Maven Multi-Module Projects. https://maven.apache.org/guides/mini/guide-multiple-modules.html
[26] Microsoft. 2019. Visual Studio C++ Project system extensibility and toolset integration – .tlog files. https://docs.microsoft.com/en-us/visualstudio/extensibility/visual-cpp-project-extensibility?view=vs-2019{#}tlog-files
[27] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 21–30. https://doi.org/10.1109/icst.2011.60
[28] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven's Test Isolation: Experience Report. In *Proceedings of the International Symposium on Software Testing and Analysis*. 249–259. https://doi.org/10.1145/3395363.3397381
[29] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928
[30] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/icse.2019.00054
[31] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262
[32] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109. https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E
[33] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *Proceedings of the International Conference on Software Engineering*. 689–699. https://doi.org/10.1109/ICSE.2017.69
[34] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031
[35] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22. https://doi.org/10.1145/3092703.3092709
[36] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern operating systems*. Pearson.
[37] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430
[38] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198
[39] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2013. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* 25, 12 (2013), 1357–1383. https://doi.org/10.1002/smr.1634
[40] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the International Conference on Software Engineering*. 430–441.