

BabelRTS: Polyglot Regression Test Selection

Gabriele Maurina^{1b}, Walter Cazzola^{1b}, and Sudipto Ghosh^{1b}

Abstract—Regression test selection (RTS) approaches reduce the number of regression tests. Current RTS approaches are typically monoglot, i.e., their implementations target a specific language. However, many subjects under test (SUT) are polyglot, i.e., they use multiple languages. Running multiple monoglot RTS approaches separately on a polyglot SUT is unsafe because tests that involve inter-language dependencies can be missed. Moreover, a new language may require completely reimplementing an RTS approach, especially if the original implementation relies on language and runtime features that are not available in the new language. We propose a new static approach called BabelRTS, which is multilingual (supports multiple languages out of the box), polyglot (analyzes SUTs written in multiple languages), and extensible (allows adding support for new languages). A key contribution is the idea of encapsulating the language-specific aspects of RTS by using patterns and actions. A pattern specifies programming language constructs used in each file that indicate dependencies to other files written in the same or a different language. An action specifies how to identify these files in the codebase. Patterns and actions can be customized to support new languages without modifying the test selection algorithm. BabelRTS is not tied to a specific language run-time system or paradigm. BabelRTS currently supports 12 languages and 5 language combinations. We evaluated BabelRTS on 142 open-source monoglot and polyglot SUTs, analyzing a total of more than two billion LOC. The performance of BabelRTS was similar to the state-of-the-art monoglot approaches on monoglot SUTs. On polyglot SUTs, BabelRTS was safer in polyglot mode and selected more tests for 60% of the commits than in monoglot mode, which missed inter-language dependencies.

Index Terms—Regression testing selection (RTS), polyglot programming, language independent RTS, multi-language testing.

I. INTRODUCTION

REGRESSION testing is a common and essential software development activity, which ensures that previously tested code does not contain software bugs as a result of change [1], [2]. Modern codebases and test suites are large. For example, Google executed around 150 million tests per day across all its projects in 2017 [3]. Since running the whole test suite

every time the codebase changes is already expensive [4], when changes occur frequently, RTS approaches must be used to select only those tests that involve the execution of changed code. For example, in 2019, Meta reported using RTS to reduce the number of tests executed per change to the tens of thousands while making tens of thousands of weekly changes [5]. Researchers have proposed several static (e.g., STARTS [6], [7], FLiRTS [8] and MaRTS [9]) and dynamic RTS approaches (e.g., Ekstazi [10], Ekstazi# [11], Clover [12] and HyRTS [13]), which statically or dynamically generate a dependency graph from the SUT to select the test cases.

Need for polyglot RTS. *Monoglot software* is written in a single language, whereas *polyglot software* is written using multiple languages. Monoglot RTS analyzes code written in a single language, while polyglot RTS analyzes code written in multiple languages. Large software projects are increasingly polyglot [14], [15], [16], and developers agree that more support is needed [5], [17]. Mayer and Bauer [18] report a mean of 5 programming languages per project, with usually one main general-purpose language and the others as domain-specific languages. Some of the most commonly used programming languages in polyglot SUTs are JavaScript, Ruby and Python [16]. Kochhar et al. [19] showed that as the number of programming languages increases, the bug proneness of code increases as well. Elsner et al. [17] describe the vastness of their IT software for transport management (IVU¹) codebase, which is polyglot (13M lines of Java and C/C++ code), and how crucial it is to do polyglot RTS to save time during development in their company. Machalica et al. [5] explain the need for polyglot RTS in their codebase at Meta.

Using a monoglot RTS approach on a polyglot SUT analyzes code written only in one language while ignoring the code written in the remaining languages and the inter-language dependencies. Applying monoglot RTS on each language independently in a polyglot SUT is not enough to make a safe selection of tests because it may miss tests that execute paths through code written in multiple languages. A key motivation behind supporting polyglot RTS is that we found over 150,000 inter-language dependencies in our experiments with 33 small/medium polyglot SUTs.

Supporting language independent RTS. Current static and dynamic RTS approaches are strongly bound to the language for which they are designed because they rely on language ecosystems (runtimes, frameworks, and tools such as parsers), which are designed to work with a single language or a small group of languages. For example, Ekstazi [10] was designed for Java

Received 19 August 2024; revised 10 January 2025; accepted 18 March 2025. Date of publication 27 March 2025; date of current version 16 May 2025. This work was supported in part by MUR Project “T-LADIES” under Grant PRIN 2020TL3X8X and in part by NSF under Grant OAC 1931363. Recommended for acceptance by T. Yue. (Corresponding author: Walter Cazzola.)

Gabriele Maurina and Sudipto Ghosh are with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523 USA (e-mail: gabriele.maurina@colostate.edu; ghosh@colostate.edu).

Walter Cazzola is with the Department of Computer Science, Università degli Studi di Milano, 20133 Milan, Italy (e-mail: cazzola@di.unimi.it).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TSE.2025.3554403>, provided by the authors

Digital Object Identifier 10.1109/TSE.2025.3554403

¹<https://www.ivu.com>.

and then adapted to work with C# [11] requiring extensive code rewriting and resulting in a completely separate tool. Another RTS tool, HyRTS [20] relies on bytecode injection by using ASM [21], an approach that works on languages running on the JVM but not others. Furthermore, the language ecosystems are in continuous evolution, and RTS approaches bound to them have to be maintained with increasing overhead as multiple languages are considered.

Moreover, relying on more abstract structures, such as ASTs and CFGs, which should be language independent, does not solve the problem. ASTs and CFGs are not as independent of the language as we think because their construction depends on the features and syntax available in the SUT language, which make them different and potentially incompatible. For example, the AST and CFG of object-oriented languages differ from those of non object-oriented ones. One can see that Python² and Erlang³ have ASTs with different elements. For these reasons, composing ASTs from different languages is complicated by the need for having a deep understanding of the two languages to properly connect nodes representing different concepts. Generating ASTs requires specialized software (e.g., parser) and the time spent on the task depends on the codebase size—it could be intractable on large cases [5]. Different parsers generate ASTs in different formats that are not compatible. Furthermore, differences in the used ASTs and CFGs entail changes to the strategies adopted by the RTS approach. These strategies need to be coded and might require a redesign and rewrite of any tool to support a new language.

Some existing tools perform multilingual static analysis and support a small set of languages. For example, MLSA [22] and Pfff⁴ support three and four languages respectively. However, they are unmaintained and often support only older versions of the languages. Moreover, they analyze the languages separately and cannot analyze the dependencies between languages.

Challenges of polyglot RTS. The implementation of a generalized approach has to work with any set of languages. Languages may differ in syntax, programming paradigm, testing frameworks, execution model (e.g., interpreted vs compiled), and runtime environment (e.g., OS vs virtual machines). A generalized approach should be independent of the whole ecosystem used by the languages.

Two languages can interact in several ways. They can be used side by side in the same source file. For example, one can use `__ASM__` in C to write assembly code, `extern(C)` to declare C functions to be used from D code, and `inline` to write code snippets in other languages in a Perl source file. They can interact at the file or library level. For example, by using the Java native interface, a file written in C/C++ can use data structures defined in another file written in Java. A Python program can load a library written in C++. Two RTS approaches that work separately with languages A and B may not work with A and B together because they may be unable to detect the

inter-language dependencies. Furthermore, as Machalica et al. [5] report, existing static RTS approaches cannot be easily extended to cross language boundaries and running dynamic approaches across language boundaries is also challenging because (1) they require language-specific infrastructure, and (2) they are impractical on a large scale.

Contribution. We present a new static RTS approach called BabelRTS. It has three features that set it apart from the existing approaches. First, it is multilingual. It supports 12 languages out of the box and works with any modern and popular programming language. Second, it is polyglot. It can analyze SUTs written in multiple languages at the same time and infer inter-language dependencies. The approach is universal and is not tied to a specific language run-time system or paradigm. Last, it is extensible. It can be customized to support new languages and language combinations. The key idea is that dependencies among the SUT files can be extracted from language constructs in the SUT source code (e.g., `import` statements, inheritance, package declarations), and that selecting tests using these dependencies makes the implementation language of the SUTs irrelevant. BabelRTS uses patterns and actions to extract file dependencies. A pattern is a function that uses a regular expression to match specific language constructs in each file that indicate dependencies to other files written in the same or a different language. An action is a function that processes a match and identifies these files in the codebase. File hashing is used to determine which SUT files changed since the last RTS iteration. Both the hash calculation and comparison are independent of the SUT language. These characteristics make BabelRTS truly polyglot. Furthermore, adopting a static approach is not a limitation. It allows us to increase independence of the approach from the programming language without getting coupled to specific ecosystems, which could change over time. Additionally, from the literature [23], [24], the improvement gained by using a dynamic approach over a static approach may be marginal.

Evaluation. We evaluated BabelRTS in an extensive study that includes the comparison with three state-of-the-art RTS tools for Java, Python and JavaScript and SUTs written in 12 languages: Java, C#, C++, Python, Erlang, Go, Ruby/JRuby, Rust, R, JavaScript, Groovy and Kotlin and some of their combinations. In total, BabelRTS was evaluated on 186 monoglot and polyglot SUTs requiring the analysis of more than 2.3 billion lines of code. This study aims to answer the following research questions:

- RQ1.** Are the time reductions, test suite reductions, and test selection by BabelRTS on monoglot SUTs on par with those of the state-of-the-art tools?
- RQ2.** What test suite reduction and time reduction does BabelRTS achieve with SUTs written in programming languages for which there are no state-of-the-art tools?
- RQ3.** Is it possible to perform polyglot RTS? If yes, how much test suite reduction does BabelRTS achieve on polyglot SUTs and how many inter-language dependencies can it find?
- RQ4.** What fault detection rate do the tests selected by BabelRTS achieve on existing monoglot fault datasets?

²<https://docs.python.org/3/library/ast.html>

³<https://www.erlang.org/doc/apps/erts/absform.html>

⁴<https://github.com/facebookarchive/pfff>

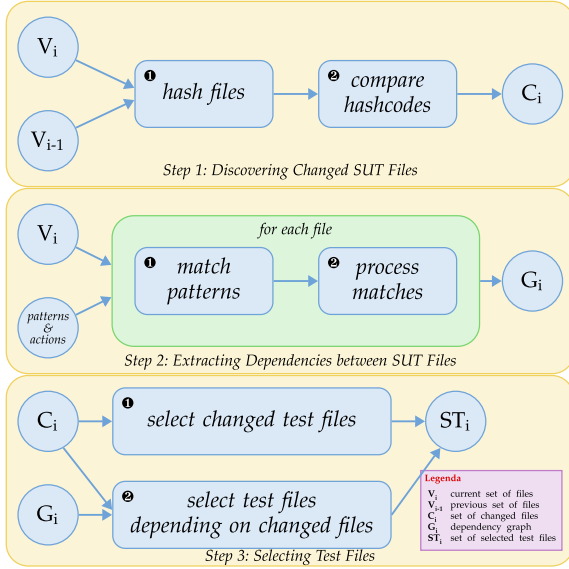


Figure 1. BabelRTS approach at the i -th iteration.

Results. The results were promising. The average recall is 93%, 74%, and 100% wrt. HyRTS [13], pytest-rts [25], and Jest. The end-to-end testing times are similar and sometimes faster than those of the considered tools. Moreover, we achieve significant time reduction and test suite reduction for both monoglot and polyglot SUTs. Additionally, we show that for 60% of the commits on polyglot SUTs, BabelRTS running in polyglot mode selected more tests that must be selected than in monoglot mode. Monoglot mode is unsafe because it misses the inter-language dependencies. Finally, BabelRTS achieves 100% fault detection on monoglot SUTs in Java, Python, and JavaScript.

Organization. This paper is organized as follows. Section II details the key ideas of BabelRTS. Section III presents the evaluation study and answers the research questions. Section IV discusses related work. Section V draws some conclusions. Appendix A describes how BabelRTS can be extended to support a new language, and Appendix B reports on the analysis we did for Java.

II. BABELRTS OVERVIEW

BabelRTS statically analyzes the SUT source code to select the test cases to be run on a new version of the SUT. Figure 1 shows the three steps of the approach: 1) discover changed files in the SUT codebase, 2) extract dependencies among SUT files and build a dependency graph, and 3) select the test files for execution. Steps 1 and 3 are independent of the SUT language because no language-specific information is needed to perform them. Only step 2 depends on the language constructs to provide monoglot and polyglot support.

To increase scalability, BabelRTS works at the file level like many recent approaches [8], [10], [11], [26]. Gligoric et al. [27] also showed that a coarser granularity can result in a reduction in overhead and a lower overall testing time. Performing

analysis at the file level enables BabelRTS to support (potentially) any language. A finer-grained approach would be more intertwined with a specific language or a family of languages, and lack flexibility to support other languages because it would analyze characteristics specific to the chosen language, which would limit the approach only to those languages that have such characteristics. For example, a class-level approach would only work on languages that have classes; a bytecode-level approach would exclude those languages such as C++ that do not compile to bytecode instead. A finer-grained approach would rely more on the language ecosystem making it harder to generalize.

A. Discovering Changes in the SUT Files

The first step is to identify the set of SUT files that changed since RTS was previously performed. When RTS is performed for the first time, all the files are treated as changed and all the test files are selected.

After the first run, BabelRTS considers two revisions: i and $i - 1$, where i is the revision ready to be tested, and $i - 1$ is the revision on which RTS was previously performed. V_i and V_{i-1} are the set of source files in i and $i - 1$ respectively. BabelRTS analyzes V_i and V_{i-1} (Figure 1, step 1.1) and determines which files changed (Figure 1, step 1.2). Given a hashing function h , and a family of functions:

$$\varphi_i : V_{i-1} \rightarrow V_i \cup \{\perp\}$$

that map a file from the older revision to the corresponding file in the new revision. $f_i \in V_i$ is considered changed if:

- $\exists f_{i-1} \in V_{i-1} \text{ s.t. } \varphi_i(f_{i-1}) = f_i \wedge h(f_{i-1}) \neq h(f_i)$; or
- $\nexists f_{i-1} \in V_{i-1} \text{ s.t. } \varphi_i(f_{i-1}) = f_i$.

The second case occurs when f_i is created after revision $i - 1$ was tested. The resulting set C_i contains all the source files of V_i that differ from the corresponding copy in V_{i-1} or are introduced in V_i . All files that existed in V_{i-1} but were eliminated before V_i are not added to C_i . This is because all the references to the eliminated files and any test case that was exercising the eliminated files should have been removed or adjusted in V_i . Moreover, C_i contains all the files that were renamed between revisions $i - 1$ and i . This is because the renaming operation is a change that should be tested. The set C_i is used in step 3 (selecting test files).

Step 1 is language independent because it uses a hashing implementation, which does not rely on language-specific knowledge. Similar approaches, such as smart checksums, have already been used by other tools [6], [10], [13]. However, BabelRTS does not use smart checksums because it requires an understanding of the SUT language to remove details irrelevant to the RTS analysis, such as comments. The smart checksum implementation used in tools such as Ekstazi, STARTS and HyRTS, operates on bytecode and not on the source files. Binding BabelRTS to bytecode would limit its applicability to languages running on the JVM. While we could substitute a more sophisticated change detection module that can handle detection of changed comments, it would make the approach less flexible because the implementation of the change detection mechanism would be tied to the SUT language.

B. Extracting Dependencies Among SUT Files

File dependency. The RTS process needs dependency information to determine which test files depend on changed source files, and thus, will exercise changed code. There is a dependency between two code portions A and B when A *uses* B and it is denoted as $A \rightarrow B$. The definition of the *use* relationship depends on the language used to implement the SUT. For example, in an object-oriented language, there is a dependency between the class C_A defined in the file A and the class C_B defined in file B when a method of C_A calls a method of C_B ; this dependency also subsumes a dependency from file A to file B. In this example, the *use* relationship is represented by the *method call* expression.

Several language constructs, not just method calls, subsume file dependencies. Let us illustrate this concept with examples. The Java statement `new Vehicle();` instantiates a new object of the class `Vehicle`. Implicitly the instantiation potentially refers to a file called `Vehicle.java` where the `Vehicle` class is defined. The inheritance statement in Java: `public class Car extends Vehicle {}` implicitly refers to the file `Vehicle.java`. For the C language, `#include "utils.h"` refers to a file called `"utils.h"`. In polyglot SUTs, constructs can also refer to files written in a different programming language. In Python, one can import a binary file compiled from C using `ctypes.CDLL('lib.so')`. In this case, the dependency is between the Python file and the C source code whose binary is archived in the `lib.so` library. Java classes can be imported in JRuby programs by using `java_import`. In this case, the Java class is contained inside a file with the same name. These and other constructs introduce file dependencies in the SUT codebase, and are common in mainstream programming languages.

Patterns and actions. BabelRTS uses patterns and actions to extract file dependencies. Patterns are functions that use regular expressions to match language constructs that refer to other files or the content of other files (possibly written in a different language). Actions are functions that process the code matched by the patterns to infer dependencies to other files. Separate patterns and actions are written for each language and language combination.

File dependencies to be inferred from a language construct may be obvious, such as names referenced by an `#include` C statement, or must be algorithmically reconstructed, such as when Java `imports` use wildcards or when loading a native library using the Python `ctypes.CDLL` command. The action task is straightforward for the former case, but for the latter, it has to collaborate with the codebase's build system to retrieve the file names. There are two ways to interact with the build system in BabelRTS: Knowing where to search for files, such as with actions that are aware of the adopted build system (e.g., knowledge of unique file structure such as `src/main/java` for source files and `src/test/java` for test files in maven), or with pattern-action pairs written for the build system files.

Dependency Graph Construction. BabelRTS uses the patterns to locate the language constructs inside the SUT files. Once a match is found (Figure 1, step 2.1), BabelRTS processes the match using the *actions* associated with the pattern (Figure 1, step 2.2). Each pattern can match several times against

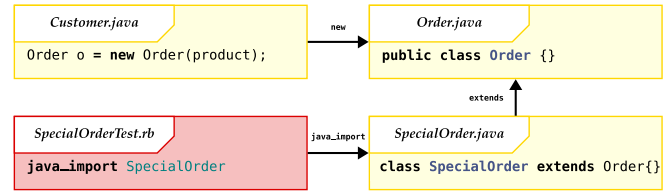


Figure 2. Dependency graph of Ruby and Java code.

Algorithm 1: Extracting the SUT dependencies.

Data: $G_i \leftarrow \{V_i, E_i = \emptyset\}$

```

1 foreach file  $\in V_i$  do
2   patterns_actions  $\leftarrow$  get_patterns_actions(file)
3   foreach pattern, action  $\in$  patterns_actions do
4     matches  $\leftarrow$  find_all(pattern, file)
5     foreach match  $\in$  matches do
6       dependencies  $\leftarrow$  action(match)
7       foreach dependency  $\in$  dependencies do
8          $E_i \leftarrow E_i \cup \{(file, dependency)\}$ 
9       end
10    end
11  end
12 end
13 return  $G_i$ 
  
```

the whole SUT codebase and each action can find zero or more dependencies to other files.

The dependencies between SUT files in iteration i result in a digraph G_i with a set of vertices, V_i , whose elements are the source and test files, and a set of edges, E_i , whose elements represent the dependencies between the files. The graph is directed because the dependencies are oriented: if file A depends on file B, it does not mean that file B depends on file A. Dependencies are treated in the same way no matter which pattern found them. The graph is simple, i.e., no edge starts and ends at the same vertex and at most one edge is present between any ordered pair of nodes in the graph. Multiple language constructs (and their respective patterns and actions) can reveal a dependency between the same pair of files, but only one edge is added to the dependency graph. The order in which the patterns are defined and file dependencies are found is irrelevant because the resulting dependency graph remains unchanged.

Figure 2 shows an example of G_i for a program written in Java and Ruby. For readability, the edges are labeled with the construct used to infer the dependency. Note that the dependency between `SpecialOrderTest.rb` and `SpecialOrder.java` is an inter-language dependency.

Algorithm 1 shows the pseudocode to create the dependency graph using patterns and actions. The function `get_patterns_actions(file)` on line 2 reads the file extension of `file` and returns the patterns and actions that must be run on files with that extension. The function `action(match)` on line 6 returns a set of files (possibly empty), which are dependencies of the file in which the match occurred. This set is used to update the dependency graph on lines 7–8.

TABLE I
SAMPLE LANGUAGE CONSTRUCTS CAPTURED BY BABELRTS

Language	Language Constructs
C#	using
C++	#include, #import
Erlang	import, load_nif, _:_(...)
Groovy	import, extends, implements, new
Java	import, package, native, extends, implements, new, _:_(...)
Javascript	import, export, require
Kotlin	import
Python	import, from _ import _, __import__
R	source
Ruby/JRuby	load, require, include, java_import
Rust	include, use

Given a language or a set of languages, multiple sets of patterns and actions can be defined in different ways to find all the file dependencies. Table I shows examples of language constructs that can be searched by BabelRTS to build the dependency graph. The languages were chosen to be representative of the mainstream programming languages and also of the main programming paradigms. The listed constructs cover most of the language dependencies but not all can be captured via patterns. Dependencies created at run-time cannot be inferred by static analysis. For example, using Java reflection, it is possible to instantiate an object whose class is dynamically determined. Even if we write a pattern that matches `Class.forName(classname)`, the file name cannot be inferred when `classname` is a variable and not a string literal. Similarly, in Python, it is possible to dynamically import a module via `__import__(module)`. This issue is a well-known limitation of static analysis [28], [29], [30], [31]. Shi et al. [24] tried to address this issue but they reported that the approaches remained unsafe in certain circumstances and they were significantly slower than the reflection-unaware alternatives.

Appendix A shows how patterns and actions can be modified and expanded to support new languages. Appendix B describes the analysis we performed for Java. The completeness of the patterns and actions for a specific language or language combination depends on the skill level of the person adding BabelRTS support for a new language. Moreover, patterns and actions can be removed or disabled when not needed by a given SUT.

Patterns and actions at work. We illustrate them using a simple codebase with both Python and C++ code files listed below, where the two languages are connected through two libraries (bar.so and foo.so).

bar.cpp	bar.h	bar.py	bar.so
bar_test.py	foo.cpp	foo.h	foo.py
foo.so	foo_test.py	makefile	

A makefile specifies which C++ code builds each library.

```
bar.so: bar.cpp bar.h
g++ -fPIC -shared -o bar.so bar.cpp

foo.so: foo.cpp foo.h
g++ -fPIC -shared -o foo.so foo.cpp
```

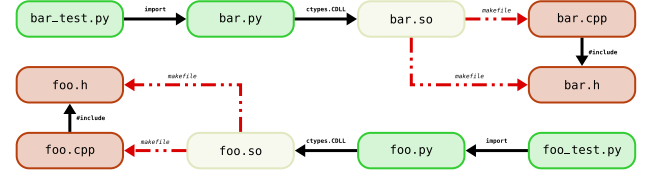


Figure 3. Dependency graph of Python and C++ code.

Figure 3 shows the dependency graph for the codebase. Black arrows are the dependencies detected using only the patterns and actions for Python and C++ without considering the information from the build system (i.e., the makefile). The labels on the arrows describe the kind of dependency.

In the absence of makefile information, BabelRTS cannot detect dependencies from Python to C++ source files because the Python and C++ patterns do not match C++ library constructs to C++ source files. However, since BabelRTS can deal with files such as libraries and build configuration files just like any other source file via patterns and actions, we can define a pattern-action pair to apply on the makefiles to determine which C++ files each library depends on.

```
S0_PATTERN = re.compile(r"(\w+\.so): (.+)")
def action(match):
    so_file, so_deps = match
    # Yield each dependency as a tuple (dependent, dependee)
    for so_dep in so_deps.split():
        yield so_file, so_dep
```

The library dependency specification inside a makefile can be matched by the above `S0_PATTERN`, which defines two capturing groups: the former captures the library name and the latter, the list of dependencies. The content of the capturing groups is passed to the action via the `match` argument. With this pattern-action pair, BabelRTS finds four additional dependencies shown as dashed red arrows in Figure 3.

C. Selecting Test Files

BabelRTS only selects those test files that exercise changed code. It uses a DFS algorithm on the digraph of dependencies built in Section II-B to find paths between test files and changed files. Given the set C_i of source files that have changed between V_{i-1} and V_i and the digraph of the latest version $G_i = \{V_i, E_i\}$, a test file $t \in V_i$ is selected if:

- $t \in C_i$; or
- $\exists v_1, v_2, \dots, v_n \in V_i$ s.t. $(t, v_1) \in E_i, v_n \in C_i, (v_j, v_{j+1}) \in E_i, 1 \leq j < n$.

The first rule selects a test file if it has changed (Figure 1, step 3.1). The second rule looks for a path, i.e., a transitive dependency, between the test file t and a changed source file v_n (Figure 1, step 3.2). This ensures that a test exercising some changed code will be selected. The set of selected test files for revision i is ST_i . ST_i is a subset of the test suite of i .

Consider the dependency graph of Figure 2. If `Order.java` changes, the test file `SpecialOrderTest.rb` is selected. If only `Customer.java` changes, the test file `SpecialOrderTest.rb` is not selected because there is no path between them. In the dependency graph of Figure 3, suppose that only `foo.h` changes.

BabelRTS does not select `bar_test.py` because no path exists in the dependency graph from it to the changed file `foo.h`. However, `foo_test.py` is selected because a path exists from it to the changed file:

`foo_test.py` $\rightarrow$ `foo.py` $\rightarrow$ `foo.so` $\rightarrow$ `foo.h`

III. EVALUATION

To answer the research questions presented in Section I, we extensively evaluated BabelRTS using three separate experiments. Experiment 1 compares the performance of BabelRTS with state-of-the-art RTS tools. Experiment 2 evaluates BabelRTS on languages for which no RTS tool exists to demonstrate that the approach works irrespective of the programming language used in the SUT. Finally, experiment 3 evaluates BabelRTS on large-scale polyglot projects to demonstrate its applicability on real-world software and the need for polyglot RTS approaches.

Experiments 1 and 2 were performed on a 64 bit machine running Fedora 35, with an Intel® Xeon® CPU E31230 at 3.20 GHz and 8 GByte of RAM. Experiment 3 was performed on a 64 bit machine running Windows 10, with an Intel® Core™ i7-11800HQ CPU at 2.30 GHz and 16 GByte of RAM.

We provided a replication package that contains all the results (including those not reported in the paper for lack of space) and the scripts needed to generate them on Zenodo.

<https://doi.org/10.5281/zenodo.14627004>.

A. Metrics

We use the following metrics in our experiments. In the definitions below, the term “*retest all*” refers to the strategy of rerunning all the tests without performing RTS.

Time reduction (TR) is computed as $\frac{T - T_{rts}}{T}$. T is the *retest all* time. T_{rts} is the *tool time*, which is the time taken by an RTS tool to perform RTS (e.g., analysis, instrumentation, graph generation, and selection) and to run the selected tests. A higher TR means that more time is saved when running the RTS tool and the selected tests, as opposed to just using the *retest all* strategy. The time reduction is negative when it takes longer to run the RTS tool and the selected tests than to use the *retest all* strategy.

Test suite reduction (TSR) is computed as $\frac{|TS| - |TS_{rts}|}{|TS|}$. TS is the entire test suite, and TS_{rts} is the set of tests selected by an RTS tool. A larger TSR is better because it leads to higher time savings. However, this depends on the execution time of the individual tests and on the selection time. Unlike TR , TSR cannot be negative because it is not possible to select more tests than those present in TS .

Reduction measurements alone cannot assess the quality of the selected tests. A tool that always selects zero tests has a perfect reduction, but is unsafe because the tests affected by the changes are not selected. On the contrary, a tool that always selects all the tests is safe, but provides no reduction.

Let TP , TN , FP , FN , P , N be the sets of *true positives*, *true negatives*, *false positives*, *false negatives*, *positives*, and *negatives*, respectively. The sets of tests selected and discarded

by a perfect tool that knows the ground truth (or the used state-of-the-art tool in the absence of ground truth) are P and N respectively. Tests selected by an RTS approach are classified as *true positive* if they should have been selected or *true negative* if they should have been discarded. Tests discarded by an RTS approach are classified as *false positive* if they should have been discarded or *false negative* if they should have been selected.

Recall is computed as $\frac{|TP|}{|P|}$, the ratio of the number of selected tests that should have been selected to the number of tests that should have been selected. It rates the selection safety, i.e., the percentage of affected tests that are selected.

Precision is computed as $\frac{|TP|}{|TP| + |FP|}$, the ratio of the number of selected tests that should have been selected to the number of selected tests. It measures the percentage of selected tests that are affected by the changes.

ILD is the number of inter-language dependencies, which refer to the dependencies between two files written in different languages in a given revision of a SUT.

Fault detection rate (FDR) is computed as $\frac{|Faults\ Detected|}{|All\ Faults|}$, the ratio of the number of detected faults to the total number of faults. It measures the percentage of detected faults, and it is one of the best indicators of safety because the FDR of a safe RTS approach should be 100%.

B. Experiment 1: State-of-the-Art Comparison

The experiment assesses BabelRTS capabilities for languages with RTS tools to demonstrate that its performance is at least on par with them. The comparison is for Java (HyRTS [13]), Python (pytest-rts [25]), and JavaScript/TypeScript (Jest⁵).

We did attempt to use Ekstazi for C++ and C# [11], BinaryRTS [32] for C++, and RTS for Erlang [33], but failures to execute them with the subjects prevented us from running the experiment. Similarly, we tried the available RTS tools for Ruby (CrystalBall⁶ and WhatToRun⁷), but they are too old and unsupported.

Since the patterns and actions used by BabelRTS on JavaScript are the same for TypeScript, we will only refer to JavaScript for the remainder of the paper.

Setup. For each language, we selected 10 SUTs from GitHub that ran on Java 8, Python 3.9, and Node 16 respectively. The SUTs selected for Java were previously used by researchers to evaluate HyRTS, Ekstazi and FLIRTS 2. For Python and JavaScript we selected SUTs that had a test suite for Pytest and Jest respectively. Table II lists the used SUTs and their average LOCs, the average size of their test suites and the average number of tests selected by each tool.

For each SUT, we selected 20 commits backward from the latest commit that compiled and ran without errors in our environment with the constraint that between two consecutive selected commits, at least 3 source files must have changed. We reversed the generated list to have the commits in chronological

⁵<https://jestjs.io>

⁶<https://github.com/toptal/crystalball>

⁷https://github.com/DyegoCosta/what_to_run

TABLE II
EXPERIMENT 1: COMPARISON WITH THE STATE-OF-THE-ART ON MONOGLOT SUTS

Java SUTs	Hash	Files	LOC	Changed	Tests	HyRTS Tests	BabelRTS Tests	Recall	Precision	HyRTS TR	BabelRTS TR	HyRTS TSR	BabelRTS TSR
commons-cli	b805e9	52.0	11233.6	5.3	28.0	1.5	26.1	1.00	0.07	-2.68	-0.02	0.95	0.07
commons-configuration	04c1bc	445.0	121911.2	4.6	165.2	22.7	47.0	0.99	0.52	0.31	0.27	0.86	0.72
commons-dbc	04e3fe	119.8	46485.0	5.3	24.5	5.3	14.1	1.00	0.61	0.14	0.39	0.88	0.69
commons-fileupload	94e5e1	59.4	10693.9	9.5	6.1	4.6	5.9	0.98	0.80	-0.00	0.45	0.50	0.32
commons-jxpath	5badb5	92.3	15973.1	12.9	20.9	16.8	9.3	0.45	0.97	-0.03	0.75	0.24	0.60
commons-lang	38cf0c	326.8	143796.2	6.4	148.9	11.2	110.4	0.99	0.11	0.50	0.12	0.93	0.26
commons-net	037644	275.6	60215.4	15.0	44.0	1.2	18.3	1.00	0.18	0.32	0.48	0.97	0.58
commons-text	59984c	138.6	33413.9	7.6	58.7	7.3	9.3	0.96	0.49	0.13	0.26	0.88	0.84
commons-validator	0924ed	148.0	33615.9	11.2	15.2	2.8	13.3	0.98	0.51	-5.91	0.37	0.59	0.22
lambda	ffb4c2	637.0	47054.8	8.0	46.0	6.3	45.1	0.92	0.59	0.02	0.91	0.46	0.08
Average		229.4	52439.3	8.6	55.7	7.9	29.9	0.93	0.49	-0.72	0.40	0.73	0.44
Python SUTs	Hash	Files	LOC	Changed	Tests	PytestRTS Tests	BabelRTS Tests	Recall	Precision	PytestRTS TR	BabelRTS TR	PytestRTS TSR	BabelRTS TSR
Augmentor	b00b72	23.2	7319.2	3.8	15.3	15.3	11.7	0.77	1.00	-0.19	0.11	0.00	0.23
backtrader	2b62e7	342.6	55208.7	37.0	81.5	81.5	73.6	0.90	1.00	-0.58	0.04	0.00	0.10
boltons	81326f	66.0	22708.6	5.6	26.6	25.7	6.0	0.26	0.99	-1.03	0.48	0.08	0.79
docker-py	19171d	138.3	30877.4	13.8	28.8	27.8	27.0	0.93	0.96	-1.16	-0.01	0.03	0.06
isort	6106d1	111.6	27204.0	5.0	33.5	33.5	27.1	0.81	1.00	-2.11	0.86	1.00	1.00
paramiko	992c99	87.0	31462.5	3.5	29.2	26.2	20.2	0.92	0.87	-1.10	0.04	0.10	0.30
pycharts	d6eba3	259.0	25357.9	51.7	48.0	48.0	46.0	0.96	1.00	-4.44	-0.07	0.00	0.04
python-prompt-toolkit	51944f	286.1	51044.2	18.5	21.0	20.0	1.5	0.07	1.00	-0.20	0.89	0.05	0.93
pytute	153317	39.5	8041.1	4.8	15.0	15.0	12.3	0.82	1.00	-0.06	0.40	0.00	0.18
tflearn	eb7238	95.6	17222.6	3.7	1.6	1.6	1.6	1.00	1.00	-2.49	0.96	1.00	1.00
Average		144.9	27644.6	14.7	30.1	29.5	22.7	0.74	0.98	-1.34	0.37	0.23	0.46
JavaScript SUTs	Hash	Files	LOC	Changed	Tests	Jest Tests	BabelRTS Tests	Recall	Precision	Jest TR	BabelRTS TR	Jest TSR	BabelRTS TSR
axios	3f53eb	147.5	21193.8	4.6	42.0	4.8	4.8	1.00	1.00	0.48	0.51	0.89	0.89
chroma.js	5df19f	145.9	23829.0	6.9	43.6	8.7	11.5	1.00	0.90	0.54	0.48	0.80	0.74
mathjs	563ff6	1020.4	124287.0	7.5	315.6	284.7	283.7	1.00	1.00	0.12	0.03	0.10	0.10
openzeppelin-contracts	217167	154.6	25180.5	6.3	82.9	11.5	13.5	1.00	0.95	0.76	0.74	0.86	0.84
react-digraph	3f853a	85.6	11695.1	6.4	20.9	11.9	14.3	1.00	0.83	0.18	0.05	0.43	0.32
riot	70461c	61.5	23774.7	8.1	11.0	4.8	4.8	1.00	1.00	0.12	0.05	0.56	0.56
substance	d32351	538.1	67145.9	6.0	41.0	2.4	2.4	1.00	1.00	0.63	0.70	0.94	0.94
taiko	578402	183.7	26067.5	5.2	63.6	35.2	37.9	1.00	0.94	0.31	0.31	0.45	0.40
videojs-contrib-hls	409c0e	49.1	27466.7	5.4	20.7	8.9	9.6	1.00	0.94	0.23	0.24	0.57	0.54
vue	810f6d	421.2	73091.4	4.0	119.0	0.7	1.0	1.00	0.83	0.74	0.81	0.99	0.99
Average		280.7	42373.1	6.0	76.0	37.3	38.3	1.00	0.94	0.41	0.39	0.66	0.63

Hash: hash of most recent used commit.

The following metrics are averages over the 20 selected revisions. **<tool>** is HyRTS for Java, python-rt for Python and Jest for Javascript.

Files: number of source and test files.

Changed: number of changed files.

LOC: number of lines of code.

Tests: total number of test files.

<tool> Tests: n. test files selected by <tool>.

Recall: BabelRTS recall wrt. <tool>.

Precision: BabelRTS precision wrt. <tool>.

<tool> TR: <tool> time reduction.

<tool> TSR: <tool> test suite reduction.

order. Starting from the oldest, we checked out each commit, ran all the tools and collected the data. We discarded the data from the starting commit because all RTS tools consider the starting commit as *new* and select all tests by default. Finally, we computed recall, precision, *TSR*, *TR* and averages per SUT. Table II reports all the collected data.

Java Results. BabelRTS recall wrt. HyRTS is close to 1 in all SUTs but one. The 95% confidence interval (CI) is 0.76 to 1.0. A high recall indicates that BabelRTS is safe wrt. HyRTS, because the tests selected by HyRTS are also selected by BabelRTS. BabelRTS precision wrt. HyRTS ranges between 0.07 and 0.97. The 95% CI is 0.13 to 0.56. A low precision means that BabelRTS selects more tests than HyRTS. This is expected, given that BabelRTS is a static tool and HyRTS is dynamic. Legunsen et al. [7] reported similar results when comparing STARTS (a static approach) with Ekstazi (a dynamic one). Additionally BabelRTS *TSR* is lower than the *TSR* of HyRTS in all SUTs but one for the same reason. However, BabelRTS *TR* is greater than that of HyRTS in eight out of ten SUTs. Furthermore, in three SUTs, HyRTS *TR* is negative, which means that it took longer to run HyRTS and its selected tests than to retest all. The average BabelRTS *TR* is 0.4 and the average HyRTS *TR* is -0.72, which means that regression testing

is faster using BabelRTS than HyRTS. The Mann-Whitney U test to compare the respective *TR* and *TSR* of the two tools gave a p-value for both as 0.0, which means that the difference is statistically significant and that while BabelRTS selects more tests, it saves more time.

Python Results. On average BabelRTS selected fewer tests than pytest-rt. BabelRTS has an average recall of 0.74 wrt. pytest-rt. The 95% CI is 0.4 to 0.99. We manually inspected ten cases in which pytest-rt selects some tests not selected by BabelRTS. In all cases, it was clear that because of a possible bug, pytest-rt selected tests that did not run changed code and should not have been selected. The average precision of BabelRTS is 0.98. The 95% CI is 0.94 to 1.0. This is expected because BabelRTS selects fewer tests. Overall BabelRTS has greater *TSR* and *TR* than pytest-rt. Our findings show that in all SUTs it took longer to run pytest-rt and its selected tests than to just use the *retest all* strategy. Thus, the *TR* of pytest-rt is negative in all SUTs. The Mann-Whitney U test for Python gave similar results as for Java. The p-value for both *TR* and *TSR* is 0, which means that the difference is statistically significant and that BabelRTS selects fewer tests and saves more time.

JavaScript Results. Both the average recall and precision of BabelRTS wrt. Jest are high: 1.0 and 0.94 respectively.

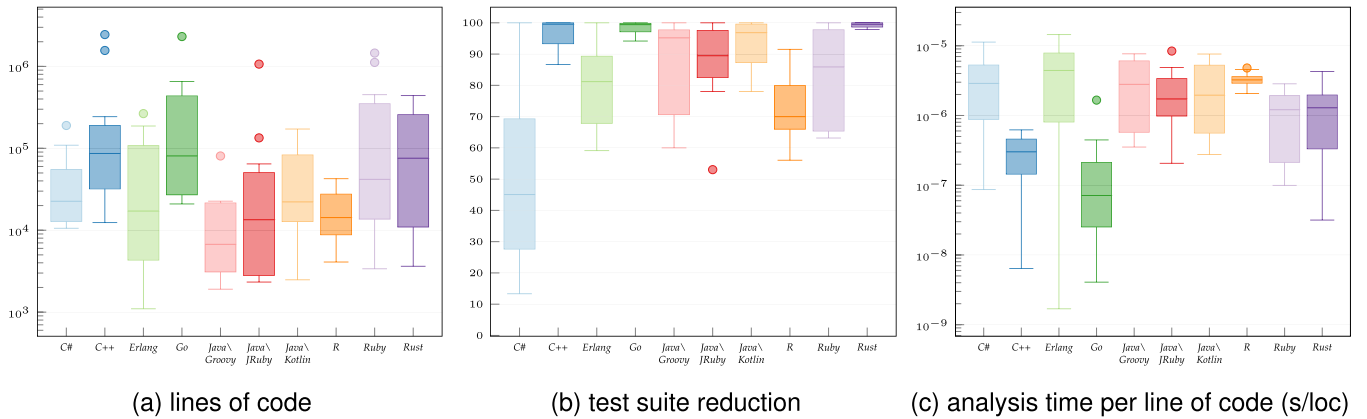


Figure 4. Report of lines of code, test suite reduction, and analysis time in experiment 2 with 10 languages.

The 95% CI is 0.99 to 1.0 and 0.83 to 0.98 respectively. Jest is a static tool like BabelRTS, and it is expected to behave similarly. Additionally, Jest uses `import` statements to build the dependency graph and thus, obtains a proper subset of the dependency graph built by BabelRTS' pattern-action pairs. The average *TSR* of the two tools are similar: 0.66 for Jest and 0.63 for BabelRTS. The average *TR* of the two tools are also similar: 0.41 for Jest and 0.39 for BabelRTS. The Mann-Whitney U test gave very different results. The p-values are 0.2 and 0.4 for *TR* and *TSR* respectively. This means that the difference is not statistically significant. This shows that BabelRTS and Jest have very similar *TR* and *TSR*.

C. Experiment 2: Languages Lacking RTS Support

The goal of this experiment is to demonstrate the ability of BabelRTS to perform RTS on monoglot and polyglot SUTs whose languages have no dedicated or usable RTS tools. Note that we cannot compare with RTS tools for C# [11], C++ [34], Erlang [33] and Ruby from the literature because they are either unavailable or we cannot run them.

Setup. We analyzed monoglot SUTs written in each of the languages, C#, C++, Erlang, Go, R, Ruby, and Rust, as well as polyglot SUTs written in pairs of languages running on the JVM. The polyglot SUTs have Java as base language, and one among Groovy, Kotlin, and Ruby as the guest language. These languages were chosen to show that BabelRTS is independent of the programming paradigm and language structures and syntax. The used language versions depend on the specific SUT analyzed. In total, we considered ten languages or language pairs. For each of them, 11 SUTs were considered for a total of 110 different SUTs. They were chosen from a variety of different domains, such as machine learning, web development and software testing, and they were developed by different communities and companies. For each of the SUTs, the last 31 consecutive commits were considered without any constraint on number of changed files, for a grand total of 3,410 revisions. The collected metrics are: *LOC*, *TSR*, analysis time, analysis time per line of code, and *ILD*. For each SUT, starting with the oldest, we ran BabelRTS on a revision, collected the considered metrics, and moved on to the next revision, 30 times in all.

We also investigated how the achieved *TSR* influenced the testing time by collecting data on *TR*. Our investigation is far from being exhaustive because some of the considered SUTs are too old or too complex to be run on our current setup. Furthermore, each SUT adopts its own development conventions as well as different testing and building systems, even within the same programming language, making it difficult to execute tests without a thorough analysis of the SUT's behavior and usage methods. This difficulty is further exacerbated by the lack of documentation on how the considered SUTs should be tested. For these reasons, we collected *TR* only for a subset of the SUTs written in C#, C++, Erlang, Ruby, Rust and Kotlin + Java. Figure 4(a) shows the average LOC span per SUT and language.

Results. BabelRTS achieves an average *TSR* of 0.54 for C# SUTs, 0.96 for C++ SUTs, 0.81 for Erlang SUTs, 0.99 for Go SUTs, 0.89 for Java-Groovy SUTs, 0.89 for Java-JRuby SUTs, 0.95 for Java-Kotlin SUTs, 0.70 for R SUTs, 0.85 for Ruby SUTs, and 0.99 for Rust SUTs. Figure 4(b) shows how *TSR* is distributed per language or language pair.

The *TSR* variation found in polyglot SUTs can be explained by differences in the languages themselves. The object-oriented paradigm might favor a better separation of concerns than pure structured programming. This might lead to separately tested independent modules, which increases the *TSR*. This is demonstrated in Section III-D where BabelRTS achieves a *TSR* of 99% on OpenJDK, a polyglot SUT with many modules independently tested.

Several factors might cause the *TSR* variation found in monoglot SUTs. Low test suite coverage makes it less likely to cover the changed code. No test needs to be selected when this occurs and the *TSR* will be larger. A large test suite might favor a larger *TSR* because more tests might not be affected by the change. A big change (where multiple source files are involved) is likely to affect more tests, which would result in a lower *TSR*. A test is more likely to be affected by a change in a dependency graph with a high edge density, thus making the *TSR* lower. For example, any change, no matter how small, will affect all the tests if the dependency graph is a clique, resulting in a zero value for *TSR*. Finally, the lack of constraints on how many changes we have between commits implies that we may

have considered empty commits, which would cause a 100% TSR, thereby increasing the average TSR.

BabelRTS achieves an average analysis time of 0.31s for C#, 0.05s for C++, 0.31s for Erlang, 0.33s for Go, 0.04s for Java-Groovy, 0.49s for Java-JRuby, 0.16s for Java-Kotlin, 0.07s for R, 0.15s for Ruby, and 0.19s for Rust SUTs. BabelRTS has a short analysis time for all the considered languages. Even a large SUT, such as Godot (written in C++) with over a million lines of code, has an average analysis time of less than 3s. The patterns and actions used or their number does not appear to meaningfully impact the analysis time. This is because the Python regular expression engine is efficient, patterns are written to be as specific as possible to make their search fast and actions are written to be computed in constant time. Figure 4(c) shows how the analysis time per line of code is distributed per language or language pair. In this case patterns and actions, and their number appear to have an impact on the analysis time per line of code, which nevertheless remains mostly below 10^{-5} seconds for all the considered languages.

For polyglot SUTs, we measured the *ILD* (defined in Section III-A). *ILD* shows the degree of interconnection of the used languages. The higher the *ILD* is, the more the involved languages interact. Polyglot RTS becomes necessary when the value of *ILD* is greater than zero. BabelRTS found an average value for *ILD* of 79; the highest value was 1,057 and the lowest was zero. Note that an *ILD* value of zero was recorded in commits where a single language was involved. The average percentages of *ILD*s over the total number of dependencies are 28%, 5%, and 24% for Groovy-Java, JRuby-Java, and Kotlin-Java, respectively.

BabelRTS achieves a positive test reduction for all the considered languages. The average *TR* values are 0.99, 0.30, 0.87, 0.59, 0.21, 0.79 for C++, C#, Erlang, Ruby, Rust, and Kotlin + Java, respectively. The overall average *TR* is 0.62. Just like with *TSR*, there is a large variation in average *TR*, which can be explained by the same factors. Additionally, we have noticed that a few outliers considerably reduced the average *TR* for the Rust language. Excluding these outliers, the average *TR* for Rust was 0.92.

D. Experiment 3: Language Interoperability

This experiment has two goals: 1) to demonstrate the scalability of BabelRTS to large codebases, and 2) to demonstrate that polyglot RTS is necessary for safe regression testing on polyglot SUTs. This experiment runs BabelRTS on OpenJDK and TensorFlow. OpenJDK is mainly written in Java with the native parts written in C++. TensorFlow is mainly written in C++ with additional bindings for other languages. We did not collect time reduction data because of the large size of the subjects and the difficulty in running them.

Setup. For each SUT, we considered 11 revisions from the main branch. The revisions are chosen so that between two consecutive revisions at least one file has changed for each of the analyzed languages. The oldest selected revision is the starting point of the RTS process. For each SUT, RTS is first performed one language at a time (monoglot) and then all languages together (polyglot). Then, the union U of the monoglot selections is computed. The collected metrics are: TSR, analysis

TABLE III
EXPERIMENT 3: SUTS STATS

	OpenJDK	TensorFlow
TSR	0.99	0.70
Analysis Time (s)	1,338	11
LOC	10,032,276	2,078,201
ILD	645	310
Dependencies	1,955,239	154,991
Files	50,691	6,965
Tests	35,880	339

time, LOC, ILD, number of dependencies, number of source and test files, and number of test files.

Results. Table III shows the metrics collected on the starting revision of OpenJDK and TensorFlow. BabelRTS analyzed ~10M lines of code in ~50K files from OpenJDK and more than 2M lines of code in almost 7K files from TensorFlow. The difference in size between the two subjects had an impact on analysis time and on TSR. The analysis time for OpenJDK is 1,338s, and the analysis time for TensorFlow is 11s. The analysis times in experiment 3 are considerably larger than the analysis times of the previous experiments (usually less than 1s) because of the large size of the considered subjects. The number of test files analyzed in OpenJDK (~35K) is two orders of magnitude larger than the number of test files analyzed in TensorFlow (339). This difference affects TSR. The average TSR is 99% on OpenJDK and 70% on TensorFlow. This is expected given that the strategy used to select the revisions produces a list of revisions whose incremental change is generally less than 10 files. Such a small change clearly makes the SUT with the largest test suite more likely to have a larger TSR. Additionally, the code in OpenJDK is structured into many modules that are tested separately, which favors a higher TSR.

BabelRTS identified ~1.9M file-level dependencies on OpenJDK and ~155K on TensorFlow. Among these dependencies, the number of inter-language dependencies is 645 and 310 respectively. These numbers show the necessity to use polyglot RTS on polyglot SUTs.

Table IV reports the results of experiment 3. The most important result is reported in the last column ($|A \setminus U|$), which shows the number of tests that are selected by the polyglot RTS, but not by the union of the selections of the corresponding monoglot RTS. Ideally, this column should be zero. Otherwise it means that the monoglot selections are missing some tests affected by the change and are therefore unsafe. The table demonstrates the lack of safety in 8 out of 10 revisions in OpenJDK and 4 out of 10 revisions in TensorFlow. This is caused by the *ILD*s that BabelRTS found in the code. This convincingly demonstrates the need for performing RTS analysis across languages. As additional evidence, we also ran BabelRTS on the most recent 1,000 commits of TensorFlow (regardless of the changes in the source code). Monoglot RTS was unsafe in ~10% of the commits, supporting the need for polyglot RTS.

E. Experiment 4: Fault Detection

The goal of this experiment is to demonstrate the fault detection ability of the tests selected by BabelRTS on existing open

TABLE IV
EXPERIMENT 3: POLYGLOT SUTs

	SHA	A	J	C	U	A∪U	SHA	A	P	C	J	U	A∪U
OpenJDK	1dc503	26486	25495	0	25495	991	1af29d	348	299	43	6	348	0
	7e0203	10	8	0	8	2	016573	302	299	3	0	302	0
	60f59a	10	8	0	8	2	43c71a	304	299	5	0	304	0
	1169a1	26487	25496	0	25496	991	321d38	305	299	6	0	305	0
	8351b3	2	1	0	1	1	a23041	317	309	8	0	317	0
	7f3250	26488	25497	0	25497	991	87a672	343	310	26	0	336	7
	95c7c5	3	3	0	3	0	c82894	318	310	0	7	317	1
	d191e4	26490	25499	0	25499	991	723288	330	311	11	2	324	6
TensorFlow	205707	2	2	0	2	0	d29f7a	317	305	4	5	314	3
	1caba0	26491	25500	0	25500	991	9daae3	325	309	15	1	325	0

A: set of tests selected when doing polyglot RTS on all languages.

J: set of tests selected when doing monoglot RTS on Java.

P: set of tests selected when doing monoglot RTS on Python.

C: set of tests selected when doing monoglot RTS on C++.

U: $J \cup P \cup C$

source fault datasets. In addition, we also demonstrate the extensibility of BabelRTS by creating custom patterns and actions to address specific dependencies that cannot be found by the base implementation of the patterns and actions in BabelRTS.

Setup. We selected three datasets from GitHub: Defects4J [35] for Java, BugsInPy [36] for Python, and BugsJS [37] for JavaScript, which contain 835, 500, and 453 faults distributed across 17, 10, and 17 SUTs respectively. For each fault, each dataset provides 1) a buggy commit V_b , 2) a fixed commit V_f , and 3) the set FT of tests that fail on V_b but pass on V_f . Two faults are never in the same commit V_b . Since FT tests fail on V_b and pass on V_f , the changes between V_b and V_f must fix the fault. Since these changes should affect the tests in FT , these tests must be selected by a safe RTS approach.

To demonstrate that BabelRTS selects the affected tests in FT , the experiment is conducted as follows. For each dataset using one fault at a time, we run BabelRTS on V_b first and then on V_f . If on the second run BabelRTS selects all tests in FT , then the fault is detected. Otherwise, the fault is not detected. We repeated this process for all the faults in all three datasets and we computed FDR for each dataset.

While running the experiment, we noticed that some faults in BugsInPy were not detected by the tests selected by BabelRTS. After a manual inspection, we discovered several issues within the dataset. We fixed all the issues that we could. For example, the dataset metadata had incorrect formats in some of the listed paths to test files (e.g., extra slash or semicolon). We removed the extra characters. Some test files were empty, but since such tests cannot fail, we removed them from FT . There was one case that we were unable to fix. Here, V_b and V_f were the same version, and thus, there was no change between them. We excluded this case in the BugsInPy dataset from our experiment.

Results. The tests selected by BabelRTS achieve a FDR of 99%, 100%, and 85% for Java, Python, and JavaScript respectively using the standard patterns and actions available out of the box in BabelRTS. Manual inspection of the missed faults helped us determine that BabelRTS was unable to detect dependencies created by external libraries rather than via the mechanisms provided by the languages. Such dependencies happened in two Java SUTs and four JavaScript SUTs out of the 44 SUTs. Using the knowledge acquired during the manual inspection, we

created custom patterns and actions to address the specific cases for each SUT. After including the new patterns and actions, the tests selected by BabelRTS achieved an FDR of 100% for all the three datasets.

In 9 out of 11 cases, creating a new pattern while reusing an existing action was sufficient. For example, some JavaScript SUTs used the libraries proxyquire and mocks, to import additional modules through a special syntax instead of the usual JavaScript statements, which would have been matched by the existing patterns. We created two new patterns to handle these cases: proxyquire\`s\*\`(\`s\*\`(.*)\`)' and load-File\`s*\`(.\*)\`/(.*)\`' respectively. The existing actions for JavaScript import can handle these two new patterns.

In 2 out of 11 cases, it was necessary to create a custom action for the new pattern. The existing pattern only handles require() and import() statements, which contain paths. For the JavaScript subject Bower, we created a new pattern, helpers\`command\`s*\`(\`s*\`(.\*)\`)' to handle helpers.command() statements, which only contain command names. Since the existing action expects a path to the file being imported, we created a new action that expects a command name and then returns a dependency to the corresponding file containing the command definition. The new action returns 'lib/commands/' + match + '.js'.

We do not include the new patterns and actions in the base implementation of BabelRTS because they are specific to the SUTs they were created for (e.g., they assume a specific folder structure, or the use of a specific library) but they are available in the replication package to enable the replication of this experiment. We are confident that the developers of an SUT would be able to create such custom patterns and actions if needed, as they have the knowledge of their own codebase.

F. Answers to Research Questions

RQ1: Based on the recall, precision, TSR , and TR , BabelRTS is on par with the state-of-the-art tools.

RQ2: BabelRTS applies to monoglot SUTs written in languages where RTS approaches are not available and achieves an average TSR of 0.86 across all the languages with a value of at least 0.6 for each language. Additionally, BabelRTS achieves

an average TR of 0.62 on the languages that it has been tested on.

RQ3: BabelRTS can scale up to large polyglot codebases. Moreover, the results demonstrate the need for polyglot RTS approaches that consider the codebase as a whole and the dependencies that exist between files of the same language and of different languages.

RQ4: The tests selected by BabelRTS achieve 100% fault detection in monoglot SUTs written in Java, Python, and JavaScript when using custom patterns and actions. Without custom patterns and actions, the selected tests achieve on average 95% fault detection on the same SUTs.

G. Threats to Validity

This subsection discusses the construct, internal, and external threats to validity of the evaluation.

Construct. The precision and recall computed in experiment 1 might be inaccurate because the other tools used to compute the ground truth might themselves be unsafe or imprecise. This is partly mitigated by repeating the experiment in multiple languages so that multiple values of precision and recall are collected and considered when discussing the performance of BabelRTS. Moreover, we used both static and dynamic tools that were developed by different people, some in research and some in industry.

Internal. BabelRTS may have bugs affecting its safety and precision. This was mitigated by writing a test suite with 100% statement coverage of the BabelRTS implementation. Another threat is that BabelRTS is not able to find all dynamic dependencies by only performing static analysis. Most notably, in Java it is possible to instantiate a class using reflection (i.e., `Class.forName()`). We have investigated this possibility and we found that, of the eleven Java SUTs that we used in the evaluation, only two of them use this kind of dynamic dependencies. Moreover, in those SUTs, the dynamic dependencies occurred in only about 1% of the files. We believe that this issue does not severely affect the usability of BabelRTS. Additionally, Shi et al. [24] demonstrated that solving this issue would not make the approach safer. Another threat relates to the completeness of the patterns and actions used for each language and the combinations. We built the patterns for each language by considering its standard definition, and for each construct in the language, we determined whether it could lead to any dependency between two code files. This method should be adequate to ensure that the patterns cover all the relevant cases. Based on our expertise with the languages, we are confident for Java, JavaScript and Python, but for the other languages it is possible that we have not considered all the corner cases.

External. The languages considered in the evaluation may not be representative of the broader set of available languages. The constructs found by BabelRTS in the considered languages might be completely missing in others and so BabelRTS might not work in the latter. This is partially mitigated by the number of programming languages considered in the evaluation and by their diversity in syntax, programming paradigms and building systems.

IV. RELATED WORK

There is extensive literature on RTS. This section describes some works whose domain partially overlaps with that of BabelRTS, including code- and model-based RTS tools, and tools for static analysis that support multiple languages. This section does not include works that are ML-based [38], because they are not directly comparable with BabelRTS.

Monoglot RTS. Several code-based RTS tools exist, some are static (e.g., STARTS [6], [7], Reks [39], Fine* [40]) and some are dynamic (e.g., Ekstazi [10], Ekstazi# [11], Clover [12] and HyRTS [13]). Some of these tools were used in our evaluation, but none of them supports multiple languages. They have to be rewritten to support a different language as happened with Ekstazi [10] and its variant Ekstazi# [11], supporting Java and C#, respectively. All these tools create a graph of dependencies, either from the source code or by executing the code, but none of them uses regular expressions like BabelRTS. Ekstazi [10], HyRTS [13] and STARTS [6] use smart checksums for the bytecode to identify changed files. On the other hand, BabelRTS directly hashes the source files because it is also designed to work with languages that do not support bytecode.

Other papers propose RTS tools for programming languages also supported by BabelRTS. Fu et al. [34] propose an approach to RTS for C++ projects that compile to LLVM IR and use the Google Test framework. Rothermel and Harrold [1] propose an approach for C that analyzes the control flow graph to find statement-level change. Bozó and Tóth [33] propose an RTS approach for Erlang using impact analysis through call graph generation via program slicing.

Polyglot RTS. Multi-language support or language independence can be pursued by moving the analysis from code to different information. SPIRITuS [41] and ReTEST [42] use test coverage and failure history data, which are not always available or computable. Briand et al. [43], Farooq et al. [44], FLiRTS [8], MaRTS [9], and FLiRTS2 [26] propose approaches based on either structural or behavioral models such as UML class, sequence and activity diagrams. Their applicability is constrained by the unavailability of the models and the difficulty of automatically extracting models from code. Their results are undermined by the coverage these abstractions provide over the codebase.

Celik et al. [45] proposed a dynamic approach that works at the operating system (OS) level to identify files outside of the JVM (i.e., not class files) that have been loaded by a particular test. Their approach can only find a test's dependency when the dependent file is loaded at run time. This can be a problem when a language runtime loads all files at once before the execution, or when all the code is compiled into a single file that is loaded once. Furthermore, their approach is inextricably bound to the OS because it is designed as a loadable Linux kernel module that captures Linux system calls to detect file dependencies. To switch to a different OS would require coping with different system calls and to use different tools to capture them.

Elsner et al. [17] proposed a hybrid approach (i.e., both static and dynamic) for Java and C/C++, which uses a combination of system call monitoring (made with `dttrace`) and JVM class

loading monitoring to derive the graph of dependencies, as well as static analysis on the C++ code to determine the relationship between source files and DLLs. The first limitation of this approach is that dtrace cannot tell apart files within an archive, such as jars and DLLs. Therefore, any dependency will be to the archive itself and not to the actual source files used to generate the archive. The second limitation is that the test traces and DLL-to-source mappings (i.e., the graph of dependencies) are computed in a separate pipeline that runs the entire codebase. The problem is that the entire codebase has to be run each time to compute the dependency graph, which prevents saving any time, or do this every once in a while (they did it once a day) and risk having an outdated graph. The third limitation of this approach is that the JVM class loading monitoring is obviously only available for languages that run on the JVM. This is important because their approach is specifically designed for a project whose main language is Java and the RTS is done on the Java test suite. If the project's main language was anything other than Java, then the entire approach would have to be redesigned. Both these approaches are tied to the environment for which they were designed and cannot be easily generalized. Adapting these two approaches to new OS and new languages would require extensive rewriting of their implementation, because different languages have different language features, interpreters, virtual machines, and file loading mechanisms. Elsner et al. [32] proposes BinaryRTS a refinement of their previous approach focusing more on dealing with cross-language links to compiled C++ binaries. BinaryRTS relies on dynamic binary instrumentation made by an external tool (DynamoRIO) to get an execution trace of the tests and ctags to get to the C++ function definition. It has been tested on C++ but it should work with any native compiled language even if to configure ctags to go back to a different language has still to be demonstrated. BinaryRTS implements a trace enabled dynamic approach that leverages on native code instrumentation that seems difficult to have it working on a different languages.

Multilingual Static Analysis. Shatnawi et al. [46] discuss five key challenges for multilingual static analysis and list a set of requirements to address them. They propose a solution for server side Java applications combined with client side web-dialects (e.g., JSP, and JSF). Lyons et al. [22] propose the *multi-lingual software analysis* architecture, which is lightweight and modular. It focuses on generating call graphs, both intra-language and inter-language. It supports only C/C++, JavaScript and Python and it is limited in the dependencies it can find (only import statements) in comparison with BabelRTS.

V. CONCLUSION AND FUTURE WORK

This paper introduces a static approach to RTS named BabelRTS, which is multilingual (i.e., supports multiple languages out of the box), polyglot (i.e., can analyze multiple languages at the same time) and extensible (i.e., support for new languages and language combinations can be added).

BabelRTS is compared to three state-of-the-art RTS tools on SUTs written in Java, Python, and JavaScript, and it is further

evaluated on 156 SUTs written in other 11 languages and their combinations. In total, the evaluation includes 186 SUTs, 8,500 revisions, over 4 million source files and more than 2.3 billion LOC. Our investigation demonstrates the need for polyglot RTS approach because monoglot RTS was shown to be unsafe on polyglot SUTs.

In the future, we will add smart checksums to increase the TSR. We are also considering how to exploit the ideas behind RTScheck [47] to demonstrate the quality of the achieved reduction when no tool exists for comparison.

ACKNOWLEDGMENT

The authors thank Abdul Matin for adding support for the R language to BabelRTS, Tyson O'Leary and Yunik Tamrakar for helping with the study of time reduction on C# and Rust, and Prof. Ann Hess and Prof. Leo Vijayarathay for consulting regarding statistical tests.

REFERENCES

- [1] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.
- [2] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection studies," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, Apr. 2001.
- [3] J. Micco, "The state of continuous integration testing @Google," *Keynote at ICST'17*. Tokyo, Japan. Piscataway, NJ, USA: IEEE Press, Mar. 2017.
- [4] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, Jul. 1993.
- [5] M. Machalica, A. Samykin, M. Porth, and S. Chandra, "Predictive test selection," in *Proc. 41st Int. Conf. Softw. Eng. in Pract. (ICSE-SEIP'19)*, Montreal, Canada, Piscataway, NJ, USA: IEEE Press, May 2018, by different people, some in pp. 91–100.
- [6] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STATic Regression Test Selection," in *Proc. ASE'17*, Piscataway, NJ, USA: IEEE Press, Oct. 2017, pp. 949–954.
- [7] 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 *Proc. FSE*. Nov. 2016, pp. 583–594.
- [8] M. Al-Refai, W. Cazzola, and S. Ghosh, "A fuzzy logic based approach for model-based regression test selection," in *Proc. MODELS'17*. Austin, TX, USA. Piscataway, NJ, USA: IEEE Press, Sep. 2017, pp. 55–62.
- [9] M. Al-Refai, S. Ghosh, and W. Cazzola, "Supporting inheritance hierarchy changes in model-based regression test selection," *Softw. Syst. Model.*, vol. 18, no. 2, pp. 937–958, 2019.
- [10] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *Proc. ICSE'15*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 713–716.
- [11] M. Vasic, Z. Parvez, A. Milicevic, and M. Grigoric, "File-Level vs. module-level regression test selection for .NET," in *Proc. FSE'17*, Paderborn, Germany. New York, NY, USA: ACM, Sep. 2017, pp. 848–853.
- [12] M. Parfianowicz, "Open Clover," 2017, [Online]. Available: <https://openclover.org>.
- [13] L. Zhang, "Hybrid regression test selection," in *Proc. ICSE'18*, Gothenburg, Sweden. Piscataway, NJ, USA: IEEE Press, May/Jun. 2018, pp. 199–209.
- [14] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools," *J. SW Eng. Res. Develop.*, vol. 5, no. 1, 2017.
- [15] Z. Mushtaq and G. Rasool, "Multilingual source code analysis," in *Proc. ICOST*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 170–175.
- [16] T. Bissyandé, F. Thing, D. Lo, L. Jiang, and L. Réveillère, "Popularity, Interoperability, and Impact of programming languages in 100,000 open source projects," in *Proc. COMPSAC'13*, Kyoto, Japan. Piscataway, NJ, USA: IEEE Press, Jul. 2013, pp. 303–312.

- [17] D. Elsner et al., “Build system aware multi-language regression test selection in continuous integration,” in *Proc. ICSE-SEIP’22*, Pittsburgh, PA, USA. Piscataway, NJ, USA: IEEE Press, May 2022, pp. 87–96.
- [18] P. Mayer and A. Bauer, “An empirical analysis of the utilization of multiple programming languages in open source projects,” in *Proc. EASE’15*, Nanjing, China. New York, NY, USA: ACM, Apr. 2015, pp. 1–10.
- [19] P. S. Kochhar, D. Wijedasa, and D. Lo, “A large scale study of multiple programming languages and code quality,” in *Proc. SANER’16*, Osaka, Japan. Piscataway, NJ, USA: IEEE Press, Mar. 2016, pp. 563–573.
- [20] L. Zhang, D. Marinov, and S. Khurshid, “Faster mutation testing inspired by test prioritization and reduction,” in *Proc. ISSTA’13*, Lugano, Switzerland. New York, NY, USA: ACM, Jul. 2013, pp. 235–245.
- [21] É. Bruneton, R. Lenglet, and T. Coupaye, “ASM: A code manipulation tool to implement adaptable systems,” *Adaptable and Extensible Component Syst.*, vol. 30, no. 19, pp. 1–12, Nov. 2002.
- [22] D. M. Lyons, A. M. Bogar, and D. Baird, “Lightweight multilingual software analysis,” in *Proc. ICSE’17*, Madrid, Spain. Rijeka, Croatia: SciTe Press, Jul. 2017, pp. 201–207.
- [23] M. K. Shin, S. Ghosh, and L. R. Vijayarath, “An empirical comparison of four Java-based regression test selection techniques,” *J. Syst. Softw.*, vol. 186, Apr. 2022, Art. no. 111174.
- [24] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, “Reflection-aware static regression test selection,” in *Proc. OOPSLA’19*, Athens, Greece. New York, NY, USA: ACM, Oct. 2019, pp. 1–29.
- [25] E. Kauhanen, J. K. Nurminen, T. Mikkonen, and M. Pashkovskiy, “Regression test selection tool for Python in continuous integration process,” in *Proc. SANER’21*, Piscataway, NJ, USA: IEEE Press, Mar. 2021, pp. 618–621.
- [26] W. Cazzola, S. Ghosh, M. Al-Refai, and G. Maurina, “Bridging the model-to-code abstraction gap with fuzzy logic in model-based regression test selection,” *Softw. Syst. Model.*, vol. 21, no. 1, pp. 207–224, Feb. 2022.
- [27] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” in *Proc. ISSTA’15*, Baltimore, MD, USA. New York, NY, USA: ACM, Jul. 2015, pp. 211–222.
- [28] E. Bodden, A. Sewe, J. Sinschek, and H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” in *Proc. ICSE’11*, Waikiki, Honolulu, Hawaii. Piscataway, NJ, USA: IEEE Press, May 2011, pp. 241–250.
- [29] Y. Li, T. Tan, Y. Sui, and J. Xue, “Self-inferencing reflection resolution for Java,” in *Proc. ECOOP’14*, LNCS 8586. Uppsala, Sweden New York, NY, USA: Springer, Jul. 2014, pp. 27–53.
- [30] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer, “More sound static handling of Java reflection,” in *Proc. APLAS’15*, LNCS 9458. Pohang, South Korea. New York, NY, USA: Springer, 2015, pp. 485–503.
- [31] D. Landman, A. Serebrenik, and J. J. Vinju, “Challenges for static analysis of Java reflection,” in *Proc. ICSE’17*, Buenos Aires, Argentina. Piscataway, NJ, USA: IEEE Press, May 2017, pp. 507–518.
- [32] D. Elsner, S. Kacianka, S. Lipp, A. Pretschner, A. Habermann, M. Graber, and S. Reimer, “BinaryRTS: Cross-language regression test selection for C++ Binaries in CI,” in *Proc. ICST’23*, Dublin, Ireland. Piscataway, NJ, USA: IEEE Press, Apr. 2023, pp. 327–338.
- [33] I. Bozó and M. Tóth, “Selecting erlang test cases using impact analysis,” in *Proc. ICNAAM’11*, AIP, Sep. 2011, pp. 802–805.
- [34] B. Fu, S. Misailovic, and M. Gligoric, “Resurgence of regression test selection for C++,” in *Proc. ICST*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 323–334.
- [35] R. Just, D. Jalali, and M. D. Ernst, “Defects4: A database of existing faults to enable controlled testing studies for Java,” in *Proc. ISSTA’14*, San José, CA, USA. New York, NY, USA: ACM, Jul. 2014, pp. 437–440.
- [36] R. Widayarsi et al., “BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies,” in *Proc. ESEC/FGSE’20*, New York, NY, USA: ACM, Nov. 2020, pp. 1556–1560.
- [37] P. Gyimesi et al., “BUGSJJS: A benchmark and taxonomy of Javascript bugs,” *Softw. Testing, Verification Rel.*, vol. 31, no. 4, Oct. 2020.
- [38] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, “Test case selection and prioritization using machine learning: A systematic literature review,” *Empirical Softw. Eng.*, vol. 27, no. 2, Mar. 2022.
- [39] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, “Towards refactoring-aware regression test selection,” in *Proc. ICSE’18*, Gothenburg, Sweden. New York, NY, USA: ACM, May 2018, pp. 233–244.
- [40] Y. Liu, J. Zhang, P. Nie, M. Gligoric, and O. Legunsen, “More precise regression testing selection via reasoning about semantics-modifying changes,” in *Proc. ISSTA’23*, Seattle, WA, USA. New York, NY, USA: ACM, Jul. 2023, pp. 664–676.
- [41] S. Romano, G. Scanniello, G. Antoniol, and A. Marchetto, “SPIRITuS: A simple information retrieval regression test selection approach,” *Inf. Softw. Technol.*, vol. 99, pp. 62–80, Jul. 2018.
- [42] M. Azizi and H. Do, “ReTEST: A cost effective test case selection technique for modern software development,” in *Proc. ISSRE’18*, Memphis, TN, USA. Piscataway, NJ, USA: IEEE Press, Oct. 2018, pp. 144–154.
- [43] L. C. Briand, Y. Labiche, and S. He, “Automating regression test selection based on UML designs,” *J. Inf. Softw. Technol.*, vol. 51, no. 1, pp. 16–30, Jan. 2009.
- [44] Q.-u.-a. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and M. Riebsch, “A model-based regression testing approach for evolving software systems with flexible tool support,” in *Proc. ECBS’10*, Oxford, UK. Piscataway, NJ, USA: IEEE Press, Mar. 2010, pp. 41–49.
- [45] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, “Regression test selection across JVM boundaries,” in *Proc. ESEC/FSE’17*, Paderborn, Germany. New York, NY, USA: ACM, Aug. 2017, pp. 809–820.
- [46] A. Shatnawi et al., “Static code analysis. of multilanguage SW system,” Jun. 2019, *arXiv:1906.00815*.
- [47] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, “A framework for checking regression test selection tools,” in *Proc. ICSE’19*, Montréal, Canada. Piscataway, NJ, USA: IEEE Press, May 2019, pp. 430–441.



Gabriele Maurina received the bachelor’s degree from the Università degli Studi di Trento, Italy, and the master’s degree from the Università degli Studi di Milano, Italy, both in computer science. He is currently working toward the Ph.D. degree in computer science with Colorado State University. He won a six months scholarship from the Università degli Studi di Milano to work on his master’s thesis at Colorado State University. His research interests include software testing and machine learning techniques.



Walter Cazzola is currently a Full Professor with the Computer Science Department, Università degli Studi di Milano, Italy, and the Chair of the ADAPT Laboratory. He designed the mChARM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is an Associate Editor for the *Journal of Computer Languages* published by Elsevier. For more information, see <https://cazzola.di.unimi.it>



Sudipto Ghosh received the Ph.D. degree in computer science from Purdue University, USA, in 2000. He is a Professor of computer science with Colorado State University, USA. His teaching and research interests include model-based software development and software testing. He is on the editorial boards of IEEE TRANSACTIONS ON RELIABILITY, INFORMATION AND SOFTWARE TECHNOLOGY, *Software Quality Journal*, and the *Journal of Software and Systems Modeling*. For more information, see <https://www.cs.colostate.edu/~ghosh>