Last Diff Analyzer: Multi-language Automated Approver for Behavior-Preserving Code Revisions

Yuxin Wang
yuxinw@uber.com
Uber Technologies, Inc.
San Francisco, California, USA

Adam Welc*
adam@mystenlabs.com
Mysten Labs
Palo Alto, California, USA

Lazaro Clapp
lazaro@uber.com
Uber Technologies, Inc.
San Francisco, California, USA

Lingchao Chen
lingchao@uber.com
Uber Technologies, Inc.
San Francisco, California, USA

ABSTRACT

Code review is a crucial step in ensuring the quality and maintainability of software systems. However, this process can be time-consuming and resource-intensive, especially in large-scale projects where a significant number of code changes are submitted every day. Fortunately, not all code changes require human reviews, as some may only contain syntactic modifications that do not alter the behavior of the system, such as format changes, variable / function renamings, and constant extractions.

In this paper, we propose a multi-language automated code approver — Last Diff Analyzer for Go and Java, which is able to detect if a reviewable incremental unit of code change (diff) contains only changes that do not modify system behavior. It is built on top of a novel multi-language static analysis framework that unifies common features of multiple languages while keeping unique language constructs separate. This makes it easy to extend to other languages such as TypeScript, Kotlin, Swift, and others. Besides skipping unnecessary code reviews, Last Diff Analyzer could be further applied to skip certain resource-intensive end-to-end (E2E) tests for auto-approved diffs for significant reduction of resource usage. We have deployed the analyzer at scale within Uber, and data collected in production shows that approximately 15% of analyzed diffs are auto-approved weekly for code reviews. Furthermore, 13.5% reduction in server node usage dedicated to E2E tests (measured by number of executed E2E tests) is observed as a result of skipping E2E tests, compared to the node usage if Last Diff Analyzer were not enabled.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

static analysis, code reviews, automated code approver

*The work was done while the author was at Uber Technologies, Inc.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613870

ACM Reference Format:
Yuxin Wang, Adam Welc, Lazaro Clapp, and Lingchao Chen. 2023. Last Diff Analyzer: Multi-language Automated Approver for Behavior-Preserving Code Revisions. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3613870

1 INTRODUCTION

Producing high quality software is challenging, particularly if its performance and reliability has a direct impact on the financials, compliance, and general quality of service of a company where this software is being developed. Consequently, in large tech companies, multiple different types of tools and processes are typically deployed to aid programmers in developing high quality software. One such tool-assisted process is code reviews — a piece of code written by a developer is not allowed to be used in production, or indeed incorporated into the main codebase, until the code has been reviewed by one or more separate developers. On the tooling side, this is typically managed by a *code review* tool which may be combined with a *code revision* system responsible for maintaining a history of the code being developed. An example of such platform is GitHub [15], which serves both as a code revision platform and a code review system, but other alternatives exist as well such as GitLab [16] or, the now discontinued, Phabricator [37].

A typical workflow when using a code review tool looks as follows. A developer (a.k.a., the author) develops a new feature, fixes a bug etc., which creates a code *revision* reflecting a change between what is currently running in production and the new code introduced by the author. A revision is itself composed of one or more code *diffs*. Initially, there is likely a single diff in the revision, encompassing all changes made by the developer on top of the latest observed state of the shared codebase. This revision is sent to a code revision system where another developer (a.k.a, the reviewer) reviews the new code to assess its correctness and quality. The reviewer may *accept* this new code revision "as-is" in which case it can be pushed to staging / production environment, though it would typically also have to go through an automated testing phase before running it in production is allowed. More often than not, however, the reviewer has some comments about the new revision that need addressing by the author before the new code can be accepted. In order to address these comments, the author creates a new diff (an incremental set of changes), on top of the existing diffs that constitute the revision. Readers familiar with GitHub may consider a diff akin to a single commit within a GitHub Pull Request [14].

The review workflow, while quite effective in discovering problems with the code being developed, is also time consuming for at least two reasons. One reason is that the reviewers (who are in most cases also developers in their own right) often spend a significant amount of time looking at other developers' code instead of working on their own features and fixes. Additionally, developers in large companies are typically busy with multiple tasks at a time, so it may take some time before a review for the new code can be performed, which creates a time overhead on the side of the author. A reduction of the reviewing burden would then be beneficial for both the authors and the reviewers, but of course such reduction should not lead to sacrificing the final code quality.

Our contribution in this space is based on an idea that, at least in some cases, additional changes requested by the reviewer may be benign from the point of view of the application behavior — for example a reviewer may suggest a more descriptive name for a variable or a function. We have realized this idea by building a tool that can automatically detect such benign changes, in which case a given code revision is automatically and immediately approved (i.e., *auto-approved*) with no human reviewer ever being involved. Please note that this strategy of detecting benign changes opens up another opportunity for improving software development efficiency — reduction of testing time. As already mentioned earlier in this section, in addition to a code review, a revision is also often subject to automated testing that consumes both time (as the author has to wait for the tests to finish) and resources (as the tests are typically ran on dedicated server machines). If the tool can determine that a given code revision is benign, the tests can be skipped leading to significant time and capacity savings.

An additional complication in building a tool for reducing overheads related to code reviewing and testing is that large tech companies typically utilize more than one programming language. Writing a separate tool for each language is certainly feasible, but a better solution which we pursued is to create a tool that can take advantage of similarities between different languages — implement analysis for code components shared between different languages only once, leaving only the language-specific parts to be implemented separately for different languages. This not only accelerates the tool development time but also makes the tool more extensible and easier to maintain in the future.

The detailed contributions of this work are as follows:

- We describe a design and implementation of an automated analysis tool named Last Diff Analyzer[1], for discovering revisions that only introduce benign code changes. Last Diff Analyzer is capable of supporting multiple programming languages (currently Go and Java) through a cross-language intermediate representation we call MAST (Meta Abstract Syntax Tree).
- We present a detailed empirical evaluation of Last Diff Analyzer's behavior in production at Uber Technologies, Inc. — we demonstrate effectiveness of the tool based on the analysis of code review data over the period of three months and show that the tool can auto-approve 15% of diffs that *were subject to analysis* (not all diffs were analyzed due to internal review requirements and simple heuristics of our code review system).
- We analyze impact of our tool on server node capacity savings resulting from the tool's potential to eliminate unnecessary container deployments for E2E tests over 6 weeks. The result indicates the tool can skip E2E tests from ~22% of *all* diffs submitted to our code review system, leading to ~13.5% server node capacity savings (measured by number of executed E2E tests), compared to previous server node usage for E2E tests.

[1]Publicly available at https://github.com/uber-research/last-diff-analyzer.

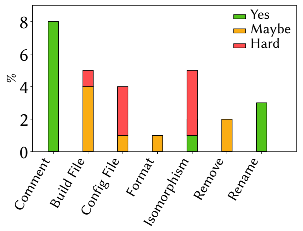
2 MOTIVATION

In order for a tool detecting benign changes to be useful in practice, and to justify its development and maintenance, it has to be able to detect significant enough number of benign changes. To determine the tool's potential we analyzed a sample of existing code diffs to see how many of them could represent benign changes, but even more importantly how many of those changes we believed could be identified by a practical tool as benign. Additionally, we later use the results of the analysis to confirm the tool's effectiveness (i.e., which benign diffs could be detected as such and which would not) in practice (Section 5.1).

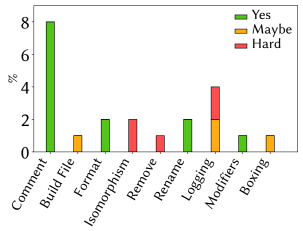
Consequently, we manually analyzed 100 diffs from our code review system for Go and Java repositories in order to:

(1) categorize code changes in each diff into several different buckets: comment, build files (e.g., dependency additions or removals), config files (YAML / JSON), format changes, isomorphism (e.g., extracting constants, extracting logic to utility functions), code removals, renamings, logging-related changes, modifier updates (Java-only), and boxing (Java-only).
(2) estimate which of the code changes could or (likely) could not be automatically detected by the tool — we split our estimates into four categories:
    (a) **yes** — this can almost certainly be auto-approved (e.g., comment changes and renaming a local variable)
    (b) **maybe** — this could be auto-approved but may require a sophisticated analysis that may be not worthwhile to implement (if such **maybe** is rarely encountered) or may be too costly to run (e.g., a more complicated change, such as re-ordering of the branches of an `if` statement)
    (c) **hard** — this could be perhaps auto-approved but in order to do so, in addition to caveats from the **maybe** category, some domain-specific knowledge that may or may not be easily available may be required (e.g., a change to a configuration file that does not affect a production run)
    (d) **no** — a change that should not be auto-approved (i.e., a change in system's behavior)

Categorization of changes was important to understand how many different "languages" we would need to support, as in addition to modifying Java and Go source files, code changes may involve modifications to build files and config files, which are often written in domain-specific (or even general) languages of their own. It was an important piece of information to collect at the initial phases of this project so that we can choose the right type of infrastructure to support all our analysis needs.



(a) Approval estimates for Go.



(b) Approval estimates for Java.

Figure 1: Approval estimates for Go and Java repositories.

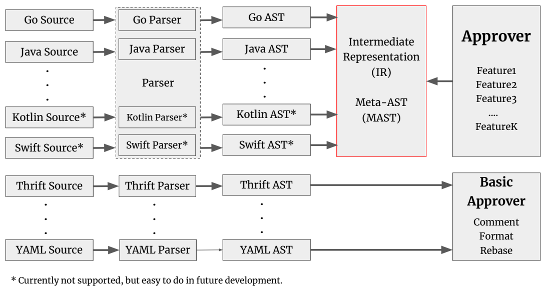


Figure 2: Architectural overview of Last Diff Analyzer.

Estimation of the number of code changes that could be auto-approved was even more important as it only makes sense to build a dedicated tool if it can provide real value to its users. The results of our estimation are shown in Figure 1a (Go) and Figure 1b (Java), and they were quite encouraging. Based on the sample data collected, we determined that 12% of Go code diffs (13% for Java) could be auto-approved with high certainty and with simple static analysis. Additional 9% of Go code diffs (4% for Java) fell into the **maybe** category which we considered the upper bound for the effectiveness of our tool (overall, 21% auto-approval rate for Go and 17% auto-approval rate for Java), with anything that we could auto-approve from the **hard** category being the proverbial cherry on top.

In Section 3 we explain how different categories of code changes are handled by our tool and in Section 5 we discuss how close our estimates were to the reality of the actual tool analyzing the same set of diffs (as well as presenting auto-approval results from running our tool in production for an extended period of time).

3 DESIGN AND IMPLEMENTATION

One of the ideas behind Last Diff Analyzer is to share the logic across different languages since many of the features (e.g., comment changes, variable / function renames, etc.) are independent of each language's unique traits. The design is then naturally split into the following parts (shown in Figure 2):

(1) convert the source files written in different languages to a unified intermediate representation — Meta Abstract Syntax Tree (MAST);
(2) implement the features of the approver *once* on top of this representation to automatically apply to all supported languages;
(3) handle the remaining language-specific features.

Note that while this strategy applies well to the languages used to write application logic (such as Go and Java) it would be unlikely to work well if we tried to, for example, unify analysis for Go and YAML, even though YAML is also considered a (serialization) language [6]. Analysis for "languages" that are not used to implement application logic is handled separately as described in Section 3.4.

3.1 Meta Abstract Syntax Tree (MAST)

3.1.1 Parsing. The first step to the unified representation is to parse the source files: converting source code to Abstract Syntax Trees (AST). Parsing is a long-solved problem and there exists many off-the-shelf parsing solutions for different languages. Here, in order to reduce the burden on the development and maintenance, the parser we choose should ideally support parsing many languages under a common architecture and a unified set of APIs to access the ASTs. We ultimately selected Tree-sitter [45] (written in plain C with no dependencies) due to its wide range of language support and friendly APIs, as well as its extensive set of language bindings, including Go which we chose to implement Last Diff Analyzer.

3.1.2 Meta AST and Translation. Although Tree-sitter provides a unified infrastructure for parsing, the AST definitions and parsing rules are developed independently and vary greatly. Consider the very similar code structures in Figure 3 written in Go and Java: even though they share a lot of similarities, the generated ASTs are noticeably different. Using tree-sitter AST as the intermediate representation directly is then infeasible considering our design goals: customized implementations of a feature may still be necessary for each language due to variations in Tree-sitter ASTs.

As a result, an additional stage in the pipeline, following the parsing step, is required to achieve further unification of the AST representations. Such unified representation should have two somewhat conflicting properties:

- **Versatility:** it should be able to fully represent a number of unique languages without losing language-specific details;
- **Unification:** it should be able to represent common structures across languages in a unified way.

To achieve this, we developed a custom AST representation called Meta Abstract Syntax Tree (MAST) for representing common AST structures in a unified format while also allowing for extensions to accommodate unique language constructs. Going back to the two similar code snippets in Figure 3, they will be represented in the exactly same MAST structure as shown in Figure 4.

If a language has a unique feature that is not shared with other languages, MAST provides the ability to "extend" its nodes via an additional field — `LangConstruct` — that is able to store additional data. This allows MAST to faithfully represent any language it supports, while keeping as many common nodes as possible for easier implementations of code analyses. The `LangConstruct` field is most useful for features that are almost the same across different languages and that yet exhibit subtle differences. Examples of such features include function declarations and field declarations, where the `LangConstruct` field can be used to extend the common MAST nodes representing these features to accommodate the slight differences each language introduces. For example, Java provides a feature to add a modifier (e.g., `public`) to a method declaration, represented as a common `FunctionDeclaration` node in MAST, to denote its visibility. In such cases, the `LangConstruct` field of the

Go:

```
func someFunc(a int, b byte) int {
    var a int
    return a + 1
}
```

```
function_declaration
  name: identifier
  parameters: parameter_list
    parameter_declaration
      name: identifier
      type: type_identifier
    parameter_declaration
      name: identifier
      type: type_identifier
  result: identifier
  body: block
    var_declaration
      var_spec
        name: identifier
        type: type_identifier
    return_statement
      expression_list
        binary_expression
          left: identifier
          right: int_literal
```

Java:

```
int someFunc(int a, byte b){
    int a;
    return a + 1;
}
```

```
method_declaration
  type: integral_type
  name: identifier
  parameters: formal_parameters
    formal_parameter
      type: integral_type
      name: identifier
    formal_parameter
      type: integral_type
      name: identifier
  body: block
    local_variable_declaration
      type: integral_type
      declarator: variable_declarator
        name: identifier
    return_statement
      binary_expression
        left: identifier
        right: decimal_integer_literal
```

Figure 3: ASTs (slightly simplified for brevity) generated by Tree-sitter for similar code written in Go and Java. Each name is a node in the tree and the underlined names are fields of the corresponding node.

`FunctionDeclaration` node representing Java methods will contain a `Modifiers` sub-field with a `public` modifier (Figure 5). Other unique features of Java method declarations, such as annotations, can be added via additional sub-fields to `LangConstruct`. Note that `LangConstruct` stores *arbitrary* data, and the `LangConstruct` field in the same node (e.g., `FunctionDeclaration`) but representing a construct in a different language will store different (or none) sub-fields to optionally accommodate other language's unique features.

For unique language elements that do not share similarities with other languages, such as `defer` (defer statements) in Go or `module` (module declarations) in Java, we create dedicated MAST nodes with language name prefix to better distinguish them.

In total, we have designed 39 common, 24 Go-specific, and 31 Java-specific MAST nodes. A set of translation components are implemented to convert Tree-sitter ASTs to MASTs for later analyses.

3.1.3 Symbolication. Having a common intermediate representation makes the implementation of multi-language code analyses easier. However, syntactical information alone is not sufficient for more sophisticated analyses. For example, in order to approve variable renamings, we need to establish links between variables and their declarations to check if all of them (and only them) have been renamed to the same new identifier.

Unfortunately, Tree-sitter does not provide typing information in the ASTs it generates. While technically we could augment MAST with full type-checking information ourselves, it was sufficient to implement a use-def analysis [1] to create "links" between each identifier node and its declaration node, and then expose the symbol information to later analysis stages. We note that this is a standard analysis; hence, we omit the details in this paper.

3.2 Unified Feature Implementation

With the unified representation (MAST), including symbol information, language-agnostic features can be implemented once to have them automatically work for all languages MAST supports

```
FunctionDeclaration
 - Name: Identifier: "someFunc"
 - Parameters:
     - ParameterDeclaration:
         - Type: Identifier: "int"
         - Name: Identifier: "a"
     - ParameterDeclaration:
         - Type: Identifier: "byte"
         - Name: Identifier: "b"
 - Body:
     - VariableDeclaration:
         - Type: Identifier: "int"
         - Name: Identifier: "a"
     - ReturnStatement:
         - BinaryExpression:
             - Identifier: "a"
             - Operator: "+"
             - IntLiteral:"1"
 - Returns:
     - ParameterDeclaration
         - Type: Identifier: "int"
```

Figure 4: Identical MAST structure for the code snippets written in Java and Go in Figure 3.

```
public int someFunc(int a, byte b){
    int a;
    return a + 1;
}
```

```
FunctionDeclaration
 - Name: ...
 - Parameters: ...
 - Body:...
 - LangConstruct
     - Modifiers: ["public"]
     - ...
```

Figure 5: The ability to store arbitrary extensions to existing MAST nodes for unique language features — Java modifiers for methods.

(currently, Go and Java). One might be tempted to implement the auto-approver by recursively comparing the MAST nodes representing the code before and after the modification via custom compare functions (indeed, this works for some classes of modifications). However, we note that some behavior-preserving modifications involve multiple MAST nodes that may be far apart in a tree: for example, renaming of a local variable affects all identifier nodes along the use-def chain. Implementing a custom compare function for an identifier node hence requires sophisticated logic that also tracks and compares the structure of the tree. Instead, Last Diff Analyzer is implemented via a series of MAST transformations. Each step in the tranformation pipeline aims at removing a certain type of the differences between two MASTs (e.g., differences introduced by variable renamings can be erased by unifying the variable names in both MASTs, which will be explained shortly). At the end of the transformations, the two MASTs are then compared for equivalence — if they are identical then the code modification was benign, otherwise the modification must be reviewed by a developer. We only resort to custom compare functions when the modification is relatively scoped (i.e., only involves the current node and its children) and a MAST transformation is not straightforward. Here, we discuss the implementation of each feature in detail.

3.2.1 Comment / Format Changes. Formatting information (i.e. locations of source-level language constructs in the source file) is automatically removed during parsing and never becomes part of MAST, so a diff containing only format changes will automatically have identical MASTs. For comments, we simply drop the comment nodes in our translation layer from Tree-sitter nodes to

MAST nodes [2]. The resulting MASTs will hence be identical if only comments or formatting are modified in a diff.

3.2.2 Unit Test File Changes. Developers write unit test files to test specific, self-contained parts of their code and ensure that it is working as intended. It is important to note that changes made to unit test files do not impact the actual behavior of the production code. Additionally, the quality of the code and tests are often guarded by other checks (such as code coverage requirements). Consequently, in Last Diff Analyzer we preprocess the set of changed files and safely ignore unit tests in our analysis.

3.2.3 Renaming. Variable and function renamings are the common requests from reviewers, often to increase maintainability of the code. Such changes do not alter the behavior of the application when compiled, and should not require re-reviews.

The idea to handle renamings is relatively straightforward: we simply need to erase the naming differences between two MASTs such that the resulting MASTs are identical. One might be tempted to disregard all identifier nodes in MASTs so the names will not affect the comparisons. However, this will lead to auto-approving changes that can modify code behavior. Consider the following example (left shows the code before the change and right shows the code after the change, differences shown in boxes):

```
1  const a = 1                        1  const a = 1
2  func myFunc(b int) bool {          2  func myFunc(b int) bool {
3    return b == 1                    3    return a == 1
4  }                                  4  }
```

Here, the only difference is in the `myFunc` body. Instead of comparing the parameter `b` and returning the comparison result, we modified the code to compare the global variable `a` instead. These obviously have different semantics, however, they will be treated as equal if Last Diff Analyzer simply ignores all identifier nodes.

Another challenge is variable shadowing (i.e., declaring a variable in inner scope that has the same name as a variable declared in outer scope). Even though the shadowing variables carry the same names as the shadowed variables, they must be treated as different objects when we erase the naming differences. For example:

```
1  func myFunc(p int) {               1  func myFunc(a_p int) {
2    if p == 2 {                      2    if a_p == 2 {
3      var p int = 3                  3      var b_p int = 3
4      if p == 3 { ··· }              4      if b_p == 3 { ··· }
5    }                                5    }
6  }                                  6  }
```

In this slightly more complex example, the name `p` is used multiple times: (1) function parameter at line 1 and its first use at line 2, then (2) local variable declaration at line 3 shadows the function parameter, and is later used at line 4. This behavior-preserving diff renames the parameter and its use with a prefix "`a_`" and the shadowing local variable `p` declared at Line 3 and its use with a prefix "`b_`". Unless shadowing is handled correctly, this code change could be considered behavior-altering if we naively compare the MASTs.

Recall that the symbol information computed during MAST construction "links" each identifier node in MAST to its corresponding declaration node, where shadowing is properly handled in the use-def analysis. For example, the first use of variable `p` at line 2 will be linked to the parameter declaration of `p` at line 1. Similarly, the use of the shadowing local variable `p` at line 4 will be linked to its declaration at line 3, instead of the parameter declaration at line 1.

With these established links, we can rename all objects (which could encompass multiple identifier nodes in a MAST) in the program according to a stable naming scheme (e.g., `OBJ_1`, `OBJ_2`, $\cdots$, where the order depends purely on the program structure) to erase the naming differences. To do this, Last Diff Analyzer first rewrites all identifiers in the declarations to be `$_OBJ_n` (illegal prefix `$` added to avoid collisions with existing identifiers) for the `n`-th declaration in a package, ordered first by the order of the files in the diff change set, relying on the diff tool [35] itself to handle file renaming, and then declarations within a file. This is illusrated conceptually as the left program below:

```
1  func $_OBJ_1($_OBJ_2 int) {        1  func $_OBJ_1($_OBJ_2 int) {
2    if p == 2 {                      2    if $_OBJ_2 == 2 {
3      var $_OBJ_3 int = 3            3      var $_OBJ_3 int = 3
4      if p == 3 { ··· }              4      if $_OBJ_3 == 3 { ··· }
5    }                                5    }
6  }                                  6  }
```

Then, in a second pass, we follow the use-def chains to update all remaining identifiers to the rewritten declaration identifiers. After such renamings, the resulting MASTs before and after the code changes will be identical (i.e., the program on the right).

If the variable at line 4 is accidentally renamed to `a_p` (shown in boxes), i.e., the use of shadowing variable is changed to use the function parameter, our naming scheme will capture this fact:

```
1  func myFunc(a_p int) {             1  func $_OBJ_1($_OBJ_2 int) {
2    if a_p == 2 {                    2    if $_OBJ_2 == 2 {
3      var b_p int = 3                3      var $_OBJ_3 int = 3
4      if a_p == 3 { ··· }            4      if $_OBJ_2 == 3 { ··· }
5    }                                5    }
6  }                                  6  }
```

Hence, the resulting MASTs will be different (one with `$_OBJ_3` and one with `$_OBJ_2` at Line 4), and this diff will be rejected.

Note that "public" (or in Go's sense, "exported") declarations that are visible across packages might be used somewhere else that are not changed by the diff, and it could break other packages if not handled properly. Therefore, Last Diff Analyzer conservatively only approves renaming private (file- or package- local) declarations.

3.2.4 Constant Additions / Removals. It is often a bad practice to leave bare literal numbers or string literals (particularly if they are used more than once) in the program since it hinders readability. Another common request from reviewers is to extract such literals to constants with more meaningful names. For example:

```
                                      1  const maxRetry = 10
1  func throttle(cnt int) bool {      2  func throttle(cnt int) bool {
2      return cnt < 10                3      return cnt < maxRetry
3  }                                  4  }
```

Here, a magic number `10` denotes the maximum number of retries for a given time period (code on the left). A bare literal number makes it hard to understand the intentions, hence a reviewer requested to add a constant value with a name `maxRetry` for better readability (code on the right). These type of changes do not affect application behavior, and in fact will be erased by compiler optimizations (constant propagation [1]).

[2]There are certain comments that carry special meanings, e.g., license headers, that should be properly reviewed if changed. Last Diff Analyzer is deployed internally where all code is considered proprietary. However, in other deployments it can be extended to preserve certain types of comment nodes during translation, which will lead to rejection if such comments are changed in the diff.

In order to auto-approve such changes, we adopt the ideas of constant propagation. Each use of the constant will be replaced with their corresponding value with the help of the symbol information, and the constant declarations will be removed from the tree. After such rewrites, the code on the right in the above example will be conceptually equivalent to the one on the left.

Similar to renamings, such rewrites are conservatively only done for file- and package- local constants.

3.2.5 Logging-Related Changes. Logging activities do not manifest themselves at the user level and it may be safe to approve some logging-related changes. In our sampled dataset, we have identified a number of diffs containing logging-related changes (e.g., changing the log levels, modifying the log messages, removing unnecessary logs etc.) that could be auto-approved.

The first step is to identify which program parts are actually responsible for logging activities. This may appear to be a simple task. However, MAST is not equipped with a type-checking algorithm (only basic use-def analysis), making it difficult to trace a function / method call to its declaration to identify logging calls. Consider the following logging call in Go (`zap` [24] is a popular logging framework in Go):

```
h.logger.Error("error!", zap.Error(err))
```

The `h` variable is an object whose `struct` type is defined in the *same* file containing a "`logger *zap.SugaredLogger`" field. Consequently, in this case we could trace the method call to its definition. However, the type information is not always available: if the type of `h` is defined in another package, we would not be able to confirm the origin of the method (for best performance and ease of implementation, our basic use-def analysis only tracks type information that exists in the modified set of files).

Due to lack of complete typing information, a heuristic is developed to identify logging calls in Go. Specifically, a method is identified to be from logging frameworks iff:

- the name is in { `Debug`, `Info`, `Warn`, `Error` }[3], *and*
- it is either (1) called with only one string argument, or (2) called with more than one argument, where the first argument is a string and one of the other arguments is a `zap` formatting function (e.g., `zap.String`, `zap.Int` etc.).

After identifying logging calls, a naive approach is to remove all of them in the program to eliminate all logging-related changes. However, an arbitrary change to a logging call can have side-effects:

(1) a function call used as an argument can change the execution of other parts of code (e.g., by inserting a piece of data into a globally available data structure)
(2) a pointer dereference (Go-only) or a method execution on null pointer value can cause a crash.

Therefore, special considerations must be taken when approving logging-related changes that contain arbitrary function calls or pointer dereferences. In summary, Last Diff Analyzer only approves the following changes to the logging calls:

(1) replacement of one logging method with another (e.g., from `h.logger.Debug` to `h.logger.Error`)
(2) removal, addition or modification of a logging call arguments that do not have side effects (e.g., literals, or functions identified using a pre-determined allowlist).

[3]We conservatively exclude `DPanic`, `Panic` and `Fatal` functions as they are rarely used and they can lead to program terminations that may change application behavior.

Due to this, Last Diff Analyzer implements this feature using cutsom compare functions. When comparing two logging call expressions, we will ignore the log levels and other side-effect-free arguments (literals or pre-determined function calls such as Go's `zap.String(...)` or Java's `String.format(...)`). Instead, we only strictly compare the other arguments (that may have side effects) in the order of their appearances. We also handle additions or removals of logging statements – Last Diff Analyzer approves a change if one tree contains a logging statement with no side effects even if the other tree does not.

For Java, the logging conventions make the identification of logging calls a little simpler: each class usually defines a `private static final logger` field, and methods inside this class will simply use the static logger throughout the execution. This guarantees that Last Diff Analyzer is able to find the type information of each logger instance, and we then apply strict checking for it (must be in an allowlist consisting of well-known types from logging frameworks). The rest of the process is the same as Go.

3.3 Language-Specific Feature Implementation

Although most of the features of Last Diff Analyzer are implemented in a unified way, there are inevitably unique language features that have to be handled separately.

One specific example is modifiers from Java (`final`, `static`, etc.) [21] which are used to denote accesses (access modifiers such as `public`, `private`, etc.) or other attributes (non-access modifiers such as `static`, `final`, etc.). Multiple modifiers (one access modifier and multiple non-access modifiers) can be added to a single declaration. However, the ordering of the modifiers does not change the semantics. Hence, when comparing two field / function declarations, the modifiers `LangConstruct` will be converted to sets to remove the ordering for comparisons.

Moreover, adding a `final` modifier to a field / function declaration imposes stricter restrictions (making the declaration non-changeable), which can be considered a "safe" change. To support this, we intentionally ignore the `final` modifier in the MAST *after* the diff when comparing against the MAST *before* the diff. Note that the reverse should not be done — removals of `final` could lead to subtle bugs that must be reviewed by developers.

3.4 Basic Approvers

A diff may contain changes to any files inside a repository with many auxiliary files written in other config / interface definition / serialization / query languages. For example, developers may submit a diff that renames a variable in the application logic written in Go, while also adding comments to the interface definitions written in Thrift™ [10]. To avoid rejecting such diffs, Last Diff Analyzer is equipped with customized basic approvers for simple features such as comments and formatting. We do not try to convert them to a unified format since (1) the features are relatively simple, and (2) these languages are arguably simpler and less similar to one another and to languages like Java and Go, compared to the similarities between one imperative general programming language to another.

3.4.1 *Build Files (Bazel™ [22]).*

The most interesting basic approver is the Bazel™ one. Bazel™ is a build system designed for large-scale multi-language multi-platform projects. While Bazel™ allow complex logic written in a subset of the Python language (see Starlark below), the majority of build configurations are written as a series of declarative "rules" that specify how source file(s) should be built (e.g., `go_library` for building a Go library) or tested (e.g., `go_test` for building a Go test suite). Most rules also provide a deps field that allows developers to specify dependent rules that must be built and linked against the current rule. From our observations in the sampled dataset (described in Section 2), there is a fair amount of diffs that remove dead dependencies or add a new dependency. Such changes are often automatically done by build file generators (such as gazelle [23]) and are considered safe from review's perspective, assuming the code still compiles (which will often be checked by continuous integration system within the code review system). Moreover, any changes to test rules are also considered safe.

Implementing such features is relatively straightforward: a custom Bazel AST comparator is implemented to (1) ignore differences between `deps` fields for selected rules (e.g., `go_library`, `java_library`, etc.) on two ASTs, and (2) ignore certain rules (e.g., `go_test`, `java_test`, etc.) altogether.

3.4.2 *Thrift™ [10], Starlark [20], SQL [25], Protobuf [19], Gomod [18], YAML [6].*

Apart from the languages discussed above, Last Diff Analyzer also offers basic support for interface definition languages such as Thrift™ and Protobuf, configuration languages such as Starlark and Gomod, serialization languages such as YAML, and data query language SQL. We note that all of these languages share very little similarities with languages for writing applications logic like Go and Java. Hence, we simply utilize off-the-shelf parsers and then rewrite their ASTs by removing the comment nodes (for approving comment changes) and position-information (for approving format changes). Finally, we compare the ASTs strictly for equality.

4 APPLICATIONS

Last Diff Analyzer is designed to work with any code review / revision system. The only required input is the source files before and after the code changes, and it will output a JSON-encoded result indicating the outcome of the analysis (Approve, Reject, or Error with an error message). This makes Last Diff Analyzer modular: it can be integrated with different code review systems, and its output can be used in various contexts. As an initial deployment, Last Diff Analyzer has been integrated with our code review system and applied to automated code reviews. We also explored applying it to skipping resource-intensive tests (e.g., E2E tests). In contrast to unit tests, which target the testing of individual functions or classes, E2E tests aim to evaluate the entirety of the application, demanding a higher allocation of resources. Section 5 shows the potential cost savings that can be achieved by skipping resource-intensive tests when there are no behavior-altering code changes in a diff.

Please note that it may depend on a specific application which diffs the Last Diff Analyzer should approve, that is which diffs are "safe", and which it should not. For example, changing a logging message may be considered "safe" for automated code-approvals. However, E2E test outcome may depend on specific log messages being present, leading to different test results if a message is modified. Due to the modular design of Last Diff Analyzer's features, we expose a battery of control flags, which independently control if a certain feature is enabled, making it possible to cater to varying requirements for different applications. Specifically, logging-related approvals are disabled for skipping E2E tests.

Another notable difference in our deployment for automated code approvals and skipping E2E tests is how we treat rebase diffs. A process of rebasing involves updating code that the diff was originally based on to the most recent version of the code found in the code repository (repository might have been modified by other developers after the diff in question was created). We consider a diff to be a rebase diff if the code updated as a result of a rebase and the code changes present in the diff do not overlap. In other words, a rebase diff does not bring any more logical changes to the existing revision. Automated code approval using Last Diff Analyzer is integrated with our code review system, which already employs heuristics to approve rebase diffs, without running Last Diff Analyzer at all. However, in skipping E2E tests, Last Diff Analyzer is separately executed by the E2E entry point script. The script decides whether to continue the test execution based on the results from Last Diff Analyzer. Under this setting, all diffs will be evaluated by Last Diff Analyzer, including rebase diffs which will be auto-approved (since they do not contain any changes).

4.1 Code Reviews

The primary practical use of Last Diff Analyzer is to facilitate automated approvals for code revisions that do not change the system's behavior. This expedites the deployment of code changes to production and, perhaps even more importantly, saves the valuable time of both authors and reviewers.

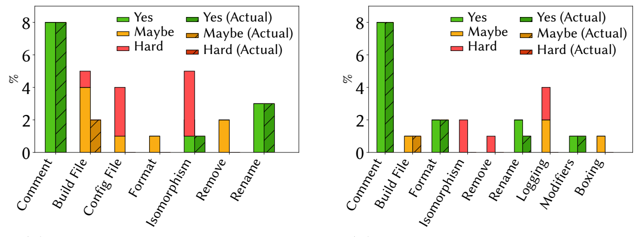
However, it is necessary to impose additional restrictions, as there may be compliance requirements to consider during the code review process that go beyond purely technical aspects. Specifically, a diff is only analyzable if the following criteria are met (otherwise Last Diff Analyzer *will not be invoked* by the code review system):

- The diff is not the very first diff in the revision;
- A required human reviewer (i.e., the owner of the code paths the revision modifies) has approved the revision before (not necessarily the immediately preceding diff – the most likely candidate here is the first diff);
- The immediately preceding diff is approved by any developer (not necessarily the code owner) or the tool itself, i.e., an approval is present between the immediately preceding diff and the current diff.

These requirements are to ensure (1) at least one required reviewer has approved this revision before, and (2) Last Diff Analyzer is only approving changes that preserve correct (previously approved) behavior of the code. We note that these are due to internal company review requirements instead of a technical choice.

4.2 Skipping Resource-Intensive Tests

E2E test pipelines are often deployment-based: a few test instances are deployed with the code changes in staging / production for each diff, and the E2E tests will be executed against the test instances. Every instance has a lifespan of up to one hour, which imposes a



(a) Actual approvals for Go. **(b) Actual approvals for Java.**

Figure 6: Actual auto-approvals for Go and Java repositories.

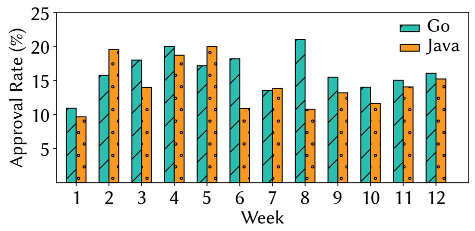


Figure 7: Weekly approval rate on Go and Java repositories.

significant demand on resources during this process. Thus, it can be inefficient to re-run the same set of E2E tests when a diff does not lead to a change in system behavior (the execution results will be the same). Therefore, in addition to automated code approvals, another application of Last Diff Analyzer is to skip E2E tests for behavior-preserving diffs. By doing this, we not only improve developer experience by reducing the time it takes for their code to be considered production-ready (all tests must pass in order for this to happen), but also free up server resources that can be better utilized elsewhere. In this application, we do not have the additional restrictions imposed by the code review system. Hence, *all* submitted diffs are subject to analysis by Last Diff Analyzer. However, Last Diff Analyzer still rejects a diff directly if it is the first diff within a revision to ensure that E2E tests are run at least once. Moreover, even if a diff is approved by Last Diff Analyzer, the E2E tests may not be skipped if the tests for the immediately preceding diff have failed. While one would expect a diff with no behavior-altering changes to fail the E2E tests whenever the previous diff fails them, this is to ensure E2E tests will be rerun if the failure is due to infrastructure problems and not the code change itself.

5 EVALUATION

To determine the effectiveness of Last Diff Analyzer, we performed an evaluation on the sampled dataset described in Section 2 (results are presented in Section 5.1). Additionally, we have deployed Last Diff Analyzer in production to both auto-approve diffs (results presented in Section 5.2) and evaluate E2E test savings (results presented in Section 5.3), and have been continuously tracking and gathering metrics to further prove its effectiveness.

5.1 Sampled Dataset

Last Diff Analyzer is first evaluated on the sampled dataset described in Section 2, consisting of 100 code revisions each on Go and Java repositories. For each code revision, all configuration flags (described in Section 4) are enabled for Last Diff Analyzer, and we collect the approval rates for each buckets we have manually categorized (**yes**, **maybe**, and **hard**).

Figure 6a shows the actual approval rates in Go repository, along with the original estimates. Last Diff Analyzer has successfully approved all **yes** code revisions, mostly from *Comment* and *Rename* features. There are two code revisions auto-approved under *Build File*, since they contain only simple dependency additions / removals, which are handled by our basic Bazel approver. The only approved revision under *Isomorphism* replaces an unused variable with "_", which is conceptually equivalent to renaming a variable.

The most interesting part lies in the rejected cases. The rejected *Build File* cases contain modifications to other aspects of Bazel build files other than dependency management, and are deemed semantically-different by Last Diff Analyzer. The rejected cases in *Config File* are obvious: they contain changes on YAML files, making it hard for Last Diff Analyzer to reason if the changes are safe. The most surprising rejection is the *Format* case. This case includes reordering of import statements, which our tool currently lacks support for. Another interesting rejected case involves replacing a deprecated API with the latest version. While technically we can approve such cases by defining a custom mapping of deprecated API with its latest version, we concluded that maintaining such a (large) list for countless libraries will be challenging, if not impossible. However, special cases may be added to Last Diff Analyzer during migration of popular common library APIs. The rest of the rejected cases may also be supported, but require more sophisticated analysis. For example, one rejected diff removes a (presumably) unused helper function. Call graphs [1] must be constructed in Last Diff Analyzer in order to approve such cases.

The results in Java repository are similar (Figure 6b). Last Diff Analyzer is able to auto-approve all but one **yes** cases (under *Rename*), due to lack of support for Scala language. The rejected cases also include removing (presumably) unused method(s), but one interesting rejected diff reorders the parameters of a method declaration. Supporting such patterns requires Last Diff Analyzer to reason beyond the scope of the method declaration to determine this change is "safe": the arguments of every call site must also be reordered accordingly. Surprisingly, Last Diff Analyzer failed to approve any of the logging-related cases. We note that this is due to our conservative logic in approving logging-related changes (Section 3.2.5): only changes to literals and side-effect-free function calls (determined by an allowlist) in the logger arguments are allowed. All rejected logging-related cases either contain custom formatting function calls, or use a custom error class for logging instead of the standard logging frameworks we support.

No false positives are observed in this evaluation. For the false negatives discussed above, we stress that the design goal is not to cover *all* possible cases, but to build an extensible multi-language foundation with maintainable logic to support most code changes. We leave more feature implementations as future work, if sufficient evidence shows a feature is a common pattern in daily development.

5.2 Code Reviews

We have deployed and enabled Last Diff Analyzer in our code review system and collected the approval rates for 12 weeks on Go and Java repositories. As shown in Figure 7, the data we collected is

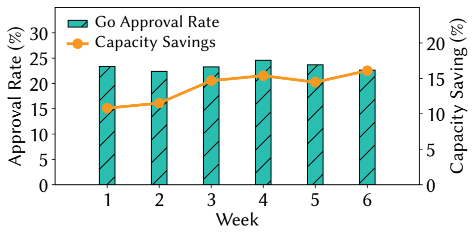


Figure 8: Weekly approval rate and capacity savings on Go repository for skipping E2E tests.

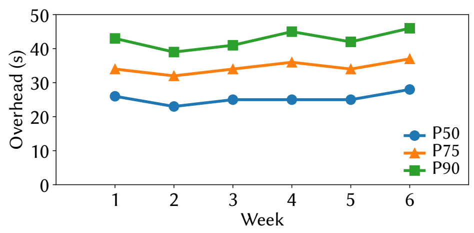


Figure 9: P50, P75, and P90 of overhead of Last Diff Analyzer over 6 weeks.

consistent with our evaluations on the sampled dataset: on average approximately 15% of the *analyzed* diffs (recall that not all diffs are subject to analysis by Last Diff Analyzer for code reviews) are auto-approved weekly for both Go and Java. We attribute the success of Last Diff Analyzer to our data-driven approach. Much of the engineering effort would have been wasted if not for the insights gathered from our sampled dataset.

5.3 Skipping Resource-Intensive Tests

We have integrated our Last Diff Analyzer in our E2E test pipelines for skipping resource-intensive tests. Before each E2E test, Last Diff Analyzer will be executed to determine if the current diff requires further testing. Due to internal deployment requirements, we are only able to collect data for 6 weeks on Go repository at the time of writing. However, we note that weekly approval rates for Java will likely be similar, as evidenced by our deployment for code reviews.

Overall, Last Diff Analyzer is able to auto-approve ~22% of all diffs, as shown in Figure 8. The approval rate is different from our deployment for code reviews (Section 5.2). However, we note that the approval rates are not comparable due to different deployment settings. For code reviews, the additional restrictions imposed by our code-hosting platform (Section 4.1) naturally excluded a number of diffs that may be auto-approved from technical perspective (e.g., rebase diffs, or diffs do not have a preceding approvals from required reviewer(s)). The approval rate is calcualted based on the diffs that are eventually *analyzed* by Last Diff Analyzer. For skipping E2E tests, such restrictions are not present and Last Diff Analyzer analyzes *all* diffs submitted to our code-hosting platform.

Note that each diff may trigger different number of E2E tests, and the E2E tests may not be skipped if the immediately preceding E2E tests failed. This leads to differences between diff approval rate and server node capacity savings. Overall, we have observed approximately 13.5% capacity savings, measured by number of executed E2E tests, relative to the node usage for E2E tests if Last Diff Analyzer were not enabled during 6 weeks of deployment.

5.4 Performance

Last Diff Analyzer requires source files before and after code revisions as input. However, our code review system only supports retrieving patch files [44] that contain code change information only, which has to be applied to unmodifed files available in the Last Diff Analyzer's execution environment. This introduces a noticeable overhead to our workflow: API calls and applying patches to a large-scale git repository is a time-consuming process.

To gain insights into the overhead of Last Diff Analyzer's integrations, we have collected the cost of running Last Diff Analyzer in our E2E pipelines for 6 weeks. Figure 9 shows the P50, P75, and P90 of the overhead for each week. A noticeable overhead of 30-60s is indeed recorded over the time period. However, it is worth mentioning that each E2E test pipeline requires hours of deployment in production. With 22% approval rate and 30s overhead, we can still expect to have a significant overall net gain. Moreover, it is possible to run Last Diff Analyzer concurrently with the tests and terminate the process early if Last Diff Analyzer approves the code revision, hence eliminating the overhead for maximum savings. It is more difficult to contrast Last Diff Analyzer's runtime overhead with the time savings and improved developer productivity when it's applied for diff auto-approval, but it is worth noting that anecdotal feedback we received from the developers after Last Diff Analyzer was deployed was very positive.

6 RELATED WORK

Significant prior research exists on detecting code refactorings. Either pure refactorings (code changes involving no change in program behavior, like the diffs Last Diff Analyzer is meant to auto-approve), or refactorings in combination with other, semantic, code changes (often across long timelines to study code evolution).

A number of AST-based analyses [9, 29, 30, 38, 42, 43, 46, 47, 49–51] have been applied to this problem (often involving the construction of complex higher-level representations such as e.g. DBs of logic facts between program entities [38] or entity-level diagrams [49, 50]). Alternatively, some approaches apply similar matching instead at the level of the program's Control Flow Graph (GFG) [3, 34]. Finally, heavier-weight static analysis methods exist for checking semantic equivalence between two programs, such as [4, 8, 17, 28, 32, 36, 39]. These methods are more robust to unknown refactorings, but at the cost of limited scalability. For example, [39] uses under-constrained symbolic execution to verify that two different functions produce the same output under each possible input, but this requires the functions in question to only perform operations that it can properly express symbolically (e.g., no floating point operations) and requires expensive queries to an SMT solver which can often timeout for complex path conditions.

In terms of applications, we also note that refactoring-detection tools have been previously turned to both the problem of assisted code review [2, 11, 12], as well as test selection [7, 48].

Additionally, there exists just as much, if not more, prior work on the related problem of code clone detection — that is, detecting similar code within the same version of a particular codebase, rather than before and after a code change (e.g. [5, 26, 27, 31, 33, 40]). This work offers some techniques which could also be applicable to the problem of detecting code refactorings.

From the long list above, we contrast three representative approaches, which are among those closest to our own:

REFDIFF 2.0. Silva et al. developed REFDIFF 2.0 [42] as multi-language refactoring detection tool based on a common AST-derived representation: Code Structure Tree (CST). REFDIFF 2.0 is focused on tracking code evolution through various refactorings, and is not concerned with identifying pure refactorings preserving program behavior. This results in two key differences between their approach and ours. First, they permit some types of refactorings (e.g. public API signature changes), which we would consider behavior altering and thus want Last Diff Analyzer to mark as semantic changes. Second, CST is designed to represent only entity-level (e.g. class, method/function) information about the codebase, while comparison between the bodies of functions is delegated to a variant of Term Frequency–Inverse Document Frequency (TF-IDF [41]). This is a generic text comparison method, based on token frequencies and using a similarity threshold, which means either syntactically small but semantically significant changes must be tolerated (for a non-zero threshold) or even identifier renamings will result in a mismatch (if the threshold is zero). In the evaluation from [42], RefDiff 2.0 exhibits 96% precision and 80% recall on a common Java refactoring detection benchmark.

RMiner Tsantalis et al.'s RMiner [46, 47] provides an AST-based statement matching algorithm that determines refactoring candidates without requiring user-defined thresholds. It uses two techniques to be resilient to code restructuring during refactoring: abstraction, which deals with changes in statements' AST node kinds due to refactorings, and argumentization, which deals with changes in sub-expressions within statements due to parameterization. Together, these techniques can detect identifier renaming and constant extraction, as well as track statements through more complex reorderings. On top of those techniques, and similar to our approach, they include particular detection rules that cover 15 representative refactoring types. Like REFDIFF 2.0, the goal is code evolution tracking, with the algorithm cascading from more to less precise statement matching rules. As presented, this matching allows for false positives and considers non-pure refactorings as true positives. Additionally, the tool as implemented is specific to Java. Thus, while some of the techniques could be used to improve Last Diff Analyzer, RMiner would not be a direct replacement.

DiffKemp Malík et al.'s DiffKemp [34], on the other hand, has a very similar objective to our own: to detect semantics-altering changes. In particular, DiffKemp has been used to check that changes between two different Linux Kernel minor versions do not alter the semantics of Kernel Application Binary Interface (KABI) functions, which are meant to be stable across the lifetime of a single major release. Like our approach, they apply some normalizing rewrites and then recursively compare code representations. But, unlike our approach, this is done at the level of CFG, rather than AST. This allows the tool to detect certain classes of refactorings which result in very different source code but identical or nearly-identical CFG (e.g. changing a *for* loop construct into a *while* loop with equivalent bounds). One immediate consequence of this is that it requires building the code before and after the change for comparison, which is significantly more expensive than our approach, requiring only parsing the changed files. Additionally, DiffKemp is implemented exclusively for C, unlike our multi-language approach.

Many of the analyses above can detect complex refactoring patterns Last Diff Analyzer does not yet support (such as outlining or inlining of fragments of function bodies), as well as more complex compositions or chains of refactorings. On the other hand, a significant number of these tools are more concerned with either understanding the history of refactorings within a codebase (ignoring behavior affecting changes interspersed within refactorings for the sake of tracing the evolution of certain parts of the code) or, conversely, focus on proving that the code before and after the change behaves *exactly* equivalently. As shown in Section 4, what constitutes a relevant change depends on the use case for the tool, and configurability of enabled features is key, in addition to generalizable infrastructure across languages.

We believe the description of Last Diff Analyzer, combined with our initial study on the kinds of diffs that would be auto-approvable, will be of interest to the broader refactoring detection and software engineering community. As will our industrial evaluation, showing the significant impact that even our initial library of refactoring patterns had in terms of saved developer time and C.I. resources.

Additionally, despite the obvious benefits, no similar tooling we are aware of enjoys broad adoption yet in the industrial setting. The closest existing tool that could potentially serve in a similar capacity to ours is Semantic [13]. It is a multi-language program analysis toolkit, also based on Tree-sitter [45], that can theoretically report semantic equivalence between two code revisions. It is not designed for situations when the code is not exactly the same but the changes are still benign. Also, unfortunately, as of the time of writing this paper (and also of the time our own project was started) the diffing functionality of Semantic is disabled [4], seemingly indefinitely.

7 CONCLUSIONS

In this paper, we have presented a multi-language AST representation called Meta Abstract Syntax Tree (MAST) that is able to represent common language features in a unified format, while allowing extensions for unique language traits, forming a foundation for multi-language program analysis. On top of this foundation, we have implemented Last Diff Analyzer— an automated code approver for detecting behavior-preserving code changes for both Go and Java. Features of Last Diff Analyzer include logging-related changes, variable renaming, constant additions / removals, and more. Evaluations from production show that Last Diff Analyzer can automatically approve up to 22% of the diffs it analyzes. When applied to skip resource-intensive E2E tests triggered by submitted diffs, Last Diff Analyzer is able to reduce ∼13.5% server node capacity usage, compared to capacity usage for E2E tests if Last Diff Analyzer were not enabled.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback, as well as everyone who contributed to this work, with special thanks to Ameya Ketkar and Raj Barik.

[4] https://github.com/github/semantic/blob/9550e30edae10b5b57598352ec4216fccf6da319/docs/examples.md#diffs

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.

[2] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 751–754. https://doi.org/10.1145/2635868.2661674

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A Differencing Algorithm for Object-Oriented Programs. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*. IEEE Computer Society, 2–13. https://doi.org/10.1109/ASE.2004.10015

[4] John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7976)*, Ezio Bartocci and C. R. Ramakrishnan (Eds.). Springer, 99–116. https://doi.org/10.1007/978-3-642-39176-7_7

[5] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*. IEEE Computer Society, 368–377. https://doi.org/10.1109/ICSM.1998.738528

[6] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008* 5 (2009), 11.

[7] Zhiyuan Chen, Hai-Feng Guo, and Myoungkyu Song. 2018. Improving regression test efficiency with an awareness of refactoring changes. *Inf. Softw. Technol.* 103 (2018), 174–187. https://doi.org/10.1016/j.infsof.2018.07.003

[8] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1027–1040. https://doi.org/10.1145/3314221.3314596

[9] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 404–428. https://doi.org/10.1007/11785477_24

[10] Apache Foundation. 2023. Thrift. https://thrift.apache.org/.

[11] Xi Ge, Saurabh Sarkar, and Emerson R. Murphy-Hill. 2014. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2014, Hyderabad, India, June 2-3, 2014*, Helen Sharp, Rafael Prikladnicki, Andrew Begel, and Cleidson R. B. de Souza (Eds.). ACM, 99–102. https://doi.org/10.1145/2593702.2593706

[12] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson R. Murphy-Hill. 2017. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*, Austin Z. Henley, Peter Rogers, and Anita Sarma (Eds.). IEEE Computer Society, 71–79. https://doi.org/10.1109/VLHCC.2017.8103453

[13] GitHub. 2022. Semantic. https://github.com/github/semantic.

[14] GitHub. 2023. About Pull Request Reviews (Documentation). https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

[15] GitHub. 2023. GitHub. https://github.com/.

[16] GitLab. 2023. GitLab. https://gitlab.com/.

[17] Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*. ACM, 466–471. https://doi.org/10.1145/1629911.1630034

[18] Google, Inc. 2023. Go Modules Reference. https://go.dev/ref/mod.

[19] Google, Inc. 2023. Protobuf. https://github.com/protocolbuffers/protobuf.

[20] Google, Inc. 2023. Starlark Language. https://github.com/bazelbuild/starlark.

[21] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional.

[22] Google Inc. 2023. Bazel. https://github.com/bazelbuild/bazel.

[23] Google Inc. 2023. Bazel Gazelle. https://github.com/bazelbuild/bazel-gazelle.

[24] Uber Technologies Inc. 2023. Zap. https://github.com/uber-go/zap.

[25] ISO/IEC. 2016. ISO/IEC 9075-1: 2016,"SQL-Part 1: Framework".

[26] J. Howard Johnson. 1993. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, October 24-28, 1993, Toronto, Ontario, Canada, 2 Volumes*, Ann Gawman, Evelyn Kidd, and Per-Åke Larson (Eds.). IBM, 171–183. https://dl.acm.org/citation.cfm?id=962305

[27] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[28] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. 2016. Relational Program Reasoning Using Compiler IR. In *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9971)*, Sandrine Blazy and Marsha Chechik (Eds.). 149–165. https://doi.org/10.1007/978-3-319-48869-1_12

[29] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 371–372. https://doi.org/10.1145/1882291.1882353

[30] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

[31] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2126)*, Patrick Cousot (Ed.). Springer, 40–56. https://doi.org/10.1007/3-540-47764-0_3

[32] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

[33] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 249–260. https://doi.org/10.1109/ICSME.2017.46

[34] Viktor Malík and Tomás Vojnar. 2021. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 329–339. https://doi.org/10.1109/ICST49551.2021.00045

[35] Michael Kerrisk. 2010. The Linux programming interface: a Linux and UNIX system programming handbook. https://man7.org/linux/man-pages/man1/diff.1.html.

[36] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, Mary Jean Harrold and Gail C. Murphy (Eds.). ACM, 226–237. https://doi.org/10.1145/1453101.1453131

[37] Phacility, Inc. 2023. Phabricator. https://www.phacility.com/phabricator/.

[38] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, Radu Marinescu, Michele Lanza, and Andrian Marcus (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[39] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 669–685. https://doi.org/10.1007/978-3-642-22110-1_55

[40] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 354–365. https://doi.org/10.1145/3236024.3236026

[41] Gerard Salton and Michael McGill. 1984. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company.

[42] Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Túlio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Trans. Software Eng.* 47, 12 (2021), 2786–2802. https://doi.org/10.1109/TSE.2020.2968072

[43] Danilo Silva and Marco Túlio Valente. 2017. RefDiff: detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 269–279. https://doi.org/10.1109/MSR.2017.14

[44] The Free Software Foundation (FSF). 2021. GNU diffutils - Comparing and Merging Files. https://www.gnu.org/software/diffutils/manual/.

[45] Tree-sitter. 2023. Tree-sitter. https://github.com/tree-sitter/tree-sitter.

[46] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.

3007722

[47] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 483–494. https://doi.org/10.1145/3180155.3180206

[48] Kaiyuan Wang, Chenguang Zhu, Ahmet Çelik, Jongwook Kim, Don S. Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 233–244. https://doi.org/10.1145/3180155.3180254

[49] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 54–65. https://doi.org/10.1145/1101908.1101919

[50] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Detection based on UMLDiff Change-Facts Queries. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*. IEEE Computer Society, 263–274. https://doi.org/10.1109/WCRE.2006.48

[51] Wuu Yang. 1991. Identifying Syntactic differences Between Two Programs. *Softw. Pract. Exp.* 21, 7 (1991), 739–755. https://doi.org/10.1002/spe.4380210706

Received 2023-05-18; accepted 2023-07-31