Test Harness Mutilation

1st Given Name Surname
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Testing is well understood to be an effective means of reducing bugs in software. But how does one prevent bugs in tests? A buggy test could lull a developer into a believing that a piece of code is correct, when it is not. To help address such problems, we introduce Necessist, a tool for finding unnecessary statements and method calls in tests. Necessist works by removing such constructs individually from a test, and then seeing whether the test passes. A test that does could contain a bug, e.g., because of an incorrect assumption held by the test's author. We give examples of new bugs found using this method. We also show that Necessist could have revealed past bugs in the Go standard library, demonstrating that Necessist is an effective tool for finding bugs in tests.**

***Index Terms*—mutation testing, multi-language tools, deterministic mutation**

I. Introduction

Testing is a fundamental part of modern software engineering [1]. Modern testing techniques automatically test complex software (such as compilers [2]), and large industry-backed infrastructure continuously runs fuzz testing on open-source projects [3]. Thus, it is not surprising when industry research suggests software testing is on the rise [4].

Given (or perhaps contributing to) this understanding of the importance of testing, command line tools for modern programming languages come with testing facilities "built in." That is, tools such as `go` for Go [5] and `cargo` for Rust [6] can automatically find, build, run, and report on the success or failure of tests within a project. Even for more mature languages such as C and C++, testing frameworks such as `CTest` [7] have seen widespread adoption.

But how does one ensure that their tests are correct? A buggy test could lull a developer into a believing that a piece of code is correct, when it is not. The situation is analogous to verifying that a piece of code implements a faulty specification (see [8, section 3.1], [9, section 5], and [10]). That is, just as a faulty specification could allow code verified against it to contain bugs, a faulty test could allow code tested with it to contain bugs.

In this paper, we introduce Necessist [11], a tool for finding certain types of bugs in tests. Necessist tries to identify statements and method calls that are *unnecessary* for a test to pass—hence, the name. Necessist does this by removing individual statements and method calls from a test—i.e., by *mutilating* the test—and then seeing whether the test passes. A test that does could contain a bug, e.g., because of an incorrect assumption held by the test's author.

Necessist implements a form of mutation testing, but it differs from conventional mutation testing tools in a few key ways. First, Necessist tries to find bugs in tests, not improve test coverage. Second, Necessist implements only one form of mutation: removal. Third, Necessist mutates code deterministically. These differences are discussed further in section I-D. But, first, we give concrete examples of bugs found by Necessist.

A. Example: Rust

Rust is a widely used, general purpose programming language [6]. The `rust-openssl` package [12] provides Rust bindings to the popular OpenSSL cryptography library [13]. An older version of a `rust-openssl` test appears in figure 1.[1] The test is meant to verify that a callback registered with `set_verify_callback` is invoked with proper arguments. In particular, the registered callback checks that a certificate is passed.

But if the callback is never invoked (e.g., because the client cannot connect to the server), the check is never performed, yet the test still passes. Necessist reveals this fact by showing that test passes with the call to `set_verify_callback` removed. One way to address this problem is to add a flag to record whether the callback is invoked. In fact, other tests within the file from which figure 1 was taken already used this approach. Following this issue's discovery, the test in figure 1 was made consistent with those other tests.

B. Example: Anchor

Anchor is a widely used framework for building and testing Solana programs [14]. `xNFTs` are executable non-fungible tokens, or "tokenized code representing ownership rights over its execution" [15], [16]. Figure 2 contains a step from one of the repository's tests.[2] The step is meant to verify that a call to a method `verify` fails under certain conditions. Notably, when the call fails, it throws an exception that is caught and subsequently ignored. Thus, the step succeeds.

Note the `assert.ok(false);` on the line following the call to `verify`. Presumably, the author expected that if the call to `verify` succeeded, the assertion would trigger and cause the test to fail. Unfortunately, the assertion similarly throws an exception that is caught by the same `catch` block.

[1] https://github.com/sfackler/rust-openssl/blob/319200ab93e252a3c0e127adc1e4c43a90f063a1/openssl/src/ssl/test/mod.rs#L114-L128

[2] https://github.com/coral-xyz/xnft/blob/1f01ea2e6cd5ab38319d018c50fb8e7359ece5c1/tests/xnft.spec.ts#L130-L135

```
#[test]
fn verify_untrusted_callback_override_ok() {
  let server = Server::builder().build();
  let mut client = server.client();
  client
    .ctx()
    .set_verify_callback(SslVerifyMode::PEER, |_, x509| {
      assert!(x509.current_cert().is_some());
      true
    });
  client.connect();
}
```

```
#[test]
fn verify_untrusted_callback_override_ok() {
  static CALLED_BACK: AtomicBool = AtomicBool::new(false);
  let server = Server::builder().build();
  let mut client = server.client();
  client
    .ctx()
    .set_verify_callback(SslVerifyMode::PEER, |_, x509| {
      CALLED_BACK.store(true, Ordering::SeqCst);
      assert!(x509.current_cert().is_some());
      true
    });
  client.connect();
  assert!(CALLED_BACK.load(Ordering::SeqCst));
}
```

Fig. 1. Problematic test in `rust-openssl` found by Necessist (top) along with its fix (bottom). The test passes with the call to `set_verify_callback` removed. Thus, if `client` cannot connect to `server`, the test passes. The fix was to add a variable `CALLED_BACK`. The test passes only if the variable is set.

```
it(
  "unless the signer does not match the curator authority",
  async () => {
    try {
      await client.verify(xnft);
      assert.ok(false);
    } catch (_err) {}
  }
);
```

```
it(
  "unless the signer does not match the curator authority",
  async () => {
    try {
      await client.verify(xnft);
      assert.ok(false);
    } catch (err) {
      const e = err as anchor.AnchorError;
      assert.strictEqual(e.error.errorCode.code,
        "CuratorMismatch");
    }
  }
);
```

Fig. 2. Problematic test in `xnft` found by Necessist (top) along with a potential fix (bottom). The test passes with the statement `await client.verify(xnft);` removed. Presumably, the author did not expect that `assert.ok(false)` would be handled by the `catch` block. A potential fix is to check the type of the exception that is caught.

Thus, the step succeeds regardless of whether the call to `verify` succeeds. Necessist reveals this fact by showing that the test passes with the call to `verify` removed. One way to address this problem is to check the type of the exception that is caught. In fact, other tests within the same file already use this approach. We submitted such a fix in a pull request to the `xNFT` repository, but have not yet received a response.

C. Implementation overview

Necessist is implemented in Rust using approximately eight KLOC (as of version 0.2.2). Table I lists the testing frameworks that Necessist currently supports, along with the frameworks' official descriptions.[3] The table also gives the *software under test (SUT) language*, i.e., the language in which the tested software is written. Note that this may be different than the language in which the tests themselves are written, called here the *testing language*. For example, both Foundry and Hardhat are used to test Solidity smart contracts. With Foundry, the tests are also written in Solidity; whereas, with Hardhat, the tests are written in TypeScript.

Supporting a testing framework requires a Rust implementation of a parser for the testing language. Table I lists the parsers used for each of the supported frameworks. This topic is discussed further in section III.

D. Comparison to conventional mutation testing

Conventional mutation testing aims to improve a project's test suite by identifying erroneous conditions the test suite fails to catch. The idea is to randomly modify the project's source code, and then see whether the project's tests still pass. Such a modified piece of source code is called a *mutant*. A mutant that causes a test to fail is said to be *killed*. A mutant for which all tests pass is said to *survive*. A project with a low *kill rate* (i.e., one for for which many mutants survive) may have an inadequate test suite.

Conventional mutation testing tools exist for most mainstream languages [25]. For the software under test (SUT) languages relevant to Necessist (see section I-C and table I), at least the following conventional mutation testing tools exist:

- C/C++: AccMut [26], c-mutate [27], Dextool [28], Mull [29], MUSIC [30], Mutate++ [31], srciror [32]
- Go: go-mutesting [33], Gremlins [34], Ooze [35]
- Rust: cargo-mutants [36], mutagen [37]
- Solidity: Deviant [38], Gambit [39], MuSC [40], RegularMutator [41], SuMo [42], Vertigo [43]

As mentioned above, Necessist implements a form of mutation testing, but it differs from conventional mutation testing tools in a few key ways.

Necessist tries to find bugs in tests, not improve test coverage. As mentioned above, conventional mutation testing tools aim to improve a project's test coverage. In this sense, conventional mutation testing tools target a project's test suite as whole, not the project's individual tests. By comparison, Necessist's aim is to find bugs in tests. Moreover, Necessist's process of removing statements and method calls does target individual tests.

Necessist performs exactly one type of mutation: removal. Conventional mutation testing tools perform random modifications of a project's source code. Such modifications are intended to resemble mistakes that a typical developer might make. Examples include changing one operator to

[3]These descriptions were taken from the projects' official webpages, and should not be construed as endorsement from the author.

Framework	Official Description	Software Under Test (SUT) Language	Testing Language	Parser
Anchor [14]	A framework for Solana's Sealevel runtime providing several convenient developer tools for writing smart contracts	C/C++/Rust	TypeScript	SWC [17]
Foundry [18], [19]	A blazing fast, portable and modular toolkit for Ethereum application development written in Rust	Solidity	Solidity	Solang [20]
Go [5]	An open source programming language that makes it easy to build simple, reliable, and efficient software	Go	Go	Tree-sitter [21]
Hardhat [22]	An Ethereum development environment for professionals	Solidity	TypeScript	SWC [17]
Rust [6], [23]	A language empowering everyone to build reliable and efficient software	Rust	Rust	syn [24]

TABLE I
SUPPORTED TESTING FRAMEWORKS

another (e.g., + to −, or < to <=), changing the value of a numeric literal, changing a use of a local variable to that of another local variable, etc. By comparison, Necessist performs only removals, specifically of statements and method calls.

Necessist mutates code deterministically. For most conventional mutation testing tools, exhausting over the set of all possible mutants would be infeasible. Thus, such tools typically test only some randomly selected subset. But, for Necessist, the types of mutations it performs are comparatively limited. Thus the set of mutants that result tends to be reasonably sized: on the order of a few thousand for a moderately sized codebase (say, a few tens of KLOC). Exhausting over a set of this size can take several hours, but is still feasible. For this reason, Necessist does not employ any sort of randomization.

E. Related work

The idea of mutating a test harness was introduced in [44, section 2.3], where it was considered in two ways. First, Groce et al. considered it as a form of optimization. Specifically, their idea was to mutate a test in a test suite, then apply conventional mutation testing, and compare the kill rate to that of the unmodified test suite. If the kill rate goes up, it could indicate that the unmodified test is sub-optimal. Second, the authors argued that most mutants of a test should reject the software under test. They made this argument by examining the mutants produced by a specific tool [27] for a specific test. In particular, no tool for mutating tests was developed. Necessist can be thought of as such a tool.

Groce et al. did not qualify *which* mutants of a test should reject the software under test, e.g., whether certain types of mutations should be more likely to result in failing tests. But if one allows *most* to mean *most mutants generated by the removal of a statement or method call*, then the authors' assertion becomes: most statements and method calls should be necessary for a test to pass. Notably, what the authors called "the only interesting surviving harness mutant" was one resulting from the removal of an assignment.

Conventional mutation testing tools were discussed in section I-D. However, one tool that stands out is Universal Mutator [45] in that it supports multiple languages like Necessist does. Unlike Necessist, however, Universal Mutator's approach to performing mutations is based on regular expressions. Thus, a Universal Mutator mutant could fail to compile because of a syntax error, for example.[4] While this approach works well for Universal Mutator, Necessist requires a more fine-grained view of each source file, which necessitates parsing.

As mentioned in section I, testing with a faulty test is analogous to verifying a program against a faulty specification. While there has been significant work on automatically detecting bugs in natural language specifications (see, e.g., [46]), to our knowledge, much less has been done on automatically detecting bugs in formal specifications. One exception is automatically detecting bugs in Alloy specifications, which has received attention within the past few years [47]–[49].

Within the realm of proof assistants, a buggy specification is effectively a buggy theorem statement. In this setting, the closest tools to Necessist we are aware of are linters for Isabelle/HOL [50] and Lean [51]. Both are capable of identifying bugs in theorem statements, e.g., unused hypotheses. However, it is not clear how techniques employed by those tools could be adapted to find bugs in tests.

The remainder of this paper is organized as follows. Section II describes our method in detail. Section III describes our implementation of the method. Section IV presents results from running Necessist on the Go standard library. Section V discusses some of the method's limitations. Section VI concludes.

II. METHOD

In this section, we describe Necessist's method for revealing bugs in tests at a high level. Generally speaking, the method should apply to most procedural languages, and testing frameworks that employ conventional idioms.

Necessist operates in two phases. In the first phase, Necessist parses each test file and identifies the *span*s (defined next) of all statements and method calls it should remove. A *span* is a data structure that names a source file, and the starting and ending locations of a piece of syntax within that source file.[5] In the second phase, Necessist removes each span, executes the associated test, and reports on the outcome.

A primary motivation for using two phases is to inform the user of progress. That is, the parsing phase is relatively quick,

[4] In principle, removing a statement or method call should not produce a syntax error, though it could produce other kinds of errors, e.g., type errors.

[5] We are unsure of where the term *span* originated, but the hypertext markup language (HTML) [52] seems plausible.

while the execution phase is comparatively slow. By knowing the total number of spans, an implementation of the execution phase can communicate how much work has been completed, and how much has yet to be done.

The two phases are described in the next two subsections.

A. Parsing

Generally speaking, Necessist will attempt to remove any statement *except* the following:

- a statement containing other statements (e.g., a `for` loop)
- a control statement (e.g., a `break`, `continue`, or `return`)
- a declaration (e.g., a local or `let` binding)
- an assertion
- the last statement in a test

The reasons for ignoring the first three classes of statements are twofold. First, removing such statements rarely result in true positives. More precisely, removing such statements tend to result in tests that fail to build, time out, fail when run, or pass for uninteresting reasons. Second, even when a passing test results, deciding whether the outcome reflects a bug tends to be more difficult than for other types of statements. For example, removing a control statement changes a test's control flow, and requires reasoning about the test in a completely different way. Thus, ignoring such statements reduces the number of results that Necessist produces, and tends to leave true positives intact.

Removing an assertion amounts to eliminating a check, which is clearly undesirable. The reason for not removing the last statement in a test is similar. Consider two cases: either the statement contains a check that could cause the test to fail, or it does not. In the former case, removing the statement would be undesirable for the same reason that removing an assertion would be undesirable. In the latter case, removing the statement would be pointless because there are no subsequent checks that could be caused to fail by the statement's removal.

Necessist's algorithm for identifying the statements it should remove appears in figure 3. The algorithm is stated as a pair of callbacks. That is, the pseudocode on lines 5-8 is meant to be called for each statement s that is not the last statement in a test. The pseudocode on lines 10-20 is meant to be called immediately after s has been processed. Note that processing s may involve executing lines 5-8 and 10-20 for other statements s'. This might occur, say, if s is a `for` loop and s' is a statement within the loop's body.

The pseudocode in figure 3 works as follows. First, three global variables are initialized (lines 1-3). The purposes of these global variables are as follows:

- $leaves\_current\_count$ holds the number of *leaf* statements (i.e., statements that do not contain other statements) that have been processed thus far.
- $count\_stack$ is a last-in-first-out data structure to hold temporary copies of the $leaves\_current\_count$ variable.
- $span\_queue$ is a first-in-first-out data structure to hold the spans of all statements and method calls that have been identified as removal candidates.

```
1: leaves_current_count ← 0
2: count_stack ← new_stack()
3: span_queue ← new_queue()

4: Before each statement s, not last in a test:
5: begin
6:     let leaves_before_count ← leaves_current_count
7:     count_stack ← push(count_stack, leaves_before_count)
8: end

9: After each statement s, not last in a test:
10: begin
11:     let leaves_before_count ← top(count_stack)
12:     count_stack ← pop(count_stack)
13:     if leaves_before_count ≠ leaves_current_count then
14:         return
15:     end if
16:     leaves_current_count ← leaves_current_count + 1
17:     if ¬is_control(s)∧¬is_declaration(s)∧¬is_ignored_call(s) then
18:         span_queue ← push(span_queue, span(s))
19:     end if
20: end
```

Fig. 3. Algorithm for identifying statements to remove

Lines 5-8 are meant to be called before each statement s that is not the last statement in a test. This part of the algorithm simply records the current value of $leaves\_current\_count$ by pushing it onto $count\_stack$.

Lines 10-20 are meant to be called immediately after s has been processed. The earlier of those lines (11-16), determine whether s is a leaf in a way that does not require listing the possible leaf types for each language. Specifically, lines 11 and 12 pop the value that was stored prior to processing s, and store the value in local variable $leaves\_before\_count$. Next, line 13 compares that value to the current value of $leaves\_current\_count$. If the two value values differ, then s is not a leaf, and line 14 returns from the callback. If the two values are equal, then s is a leaf, and line 16 increments $leaves\_current\_count$ to indicate so.

The remainder of the second callback works as follows. Line 17 checks whether s is a control statement, a declaration, or an ignored call (e.g., an assertion). It does so using three functions, is_control, is_declaration, and is_ignored_call, whose implementations are framework specific. If s does not fall into any of these categories, then line 18 pushes s's span onto $span\_queue$ for later processing.

Given the prevalence of the builder pattern [53], Necessist removes method calls in addition to statements.[6] Figure 3 gives only pseudocode for identifying statements, not for identifying method calls. Given a syntax tree for a test file, identifying method calls to be removed is relatively straightforward. There is one minor complication, however. Consider a statement that is *just* a method call, e.g.:

```
receiver.method();
```

[6]Recall, a *builder* for a type T is itself a type with one or more methods to modify the builder's internal state, and a method to ultimately construct an instance of T based on the builder's internal state. Builders are useful when the type T provides many different options for its construction, but supporting them all in one function would be unwieldy. In Rust, `std::process::Command` [54] is a type that implements the builder pattern, and that is used frequently in tests.

```
1: test_file_span_map ← group_spans_by_test_file(span_queue)
2: span_outcome_map ← new_map()
3: while test_file ← keys(test_file_span_map) do
4:     if run(test_file) = FAILED then
5:         emit_warning(test_file)
6:         continue
7:     end if
8:     while span ← test_file_span_map[test_file] do
9:         let backup ← backup(test_file)
10:        remove_span(test_file, span)
11:        if build(test_file) = FAILED then
12:            span_outcome_map[span] ← NONBUILDABLE
13:            restore(backup)
14:            continue
15:        end if
16:        span_outcome_map[span] ← run(test_file)
17:        restore(backup)
18:    end while
19: end while
```

Fig. 4. Algorithm for removing spans

Clearly, it would not make sense to remove both the statement `receiver.method();` and the method call `.method()`. At present, we remove the statement, though we do not see a clear advantage to choosing one approach over the other.

For method calls identified for removal, their spans are pushed onto the same queue as for statements, i.e., *span_queue*, which is initialized on line 3 of figure 3.

B. Execution

Once the spans of all statements and method calls have been identified, Necessist can begin removing them. The output of this phase is, for each span, one of the following four possible outcomes:

- NONBUILDABLE: The test did not build.
- TIMEDOUT: The test built, but timed out when run.
- FAILED: The test built and ran to completion, but failed.
- PASSED: The test built, ran to completion, and passed.

Note that only PASSED outcomes can indicate bugs. So, in a certain sense, distinguishing the other outcomes is unnecessary. Our reason for distinguishing them is similar to our reason for using two phases. In a protracted run, it is helpful to know that Necessist is doing useful work, even when that work is not producing passing tests. The additional outcomes help in understanding such lengthy runs.

Necessist's algorithm for removing spans appears in figure 4, and is described in the next several paragraphs.

The first step is to group the spans in *span_queue* by test file, so as to produce a map from test files to the spans that refer to them (*test_file_span_map* on line 1). Recall, *span_queue* holds the spans (i.e., source location information) of each statement and method call identified in the parsing phase. Following the creation of *test_file_span_map*, an additional map is created to hold the outcome that will be associated with each span (line 2).

Lines 3-19 consist of two nested loops. The outer loop iterates over each of the test files in the keys of *test_file_span_map*. For each such *test_file*, the first step is to run *test_file*'s tests unaltered (line 4). This step is not strictly necessary, but it helps to increase confidence in the results. For example, if a statement's removal appears to cause a test to fail, it is helpful to know that the test would pass if the statement were left in place. If the tests fail unaltered, a warning is emitted (line 5), and the outer loop proceeds to the next test file.

The inner loop on lines 8-18 iterates over each of the spans associate with *test_file*. Prior to removing the span, a backup of the test file is created (line 9). The backup is used to restore the test file at the end of the inner loop iteration (lines 13 and 17). Following the backup's creation, the span is removed on line 10.

Lines 11-15 attempt to build the test file. If the build fails, the span's outcome is set to NONBUILDABLE (line 12), the test file is restored (line 13), and the inner loop proceeds to the next iteration.

If the test file can be built with the span removed, the test file is then run (line 16). This results in one of the three possible outcomes, TIMEDOUT, FAILED, or PASSED, and the span's outcome is set accordingly. Finally, the test file is restored (line 17), and the inner loop proceeds to the next iteration.

III. IMPLEMENTATION

This section describes our implementation of the method introduced in the previous section. The description applies to Necessist version 0.2.2.

Necessist interacts with each testing framework through two Rust traits, `ParseHigh` and `RunHigh`. These are *high-level* traits for which there are corresponding *low-level* traits, `ParseLow` and `RunLow`. Intuitively, the high-level traits consist of just a few methods that each do a lot, whereas the low-level traits consist of many methods that each do a little. Furthermore, an implementation of either low-level trait can be used to implement the corresponding high-level trait.

All five of Necessist's currently supported frameworks implement `ParseHigh` via `ParseLow`. For the `Run` traits, three implement `RunHigh` via `RunLow`, and two implement `RunHigh` directly. The next two subsections provide additional details.

A. Parsing

The `ParseHigh` trait consists of just a single method that visits all of a project's test files and returns a collection of spans (essentially, *span_queue* from figure 3). As mentioned above, the `ParseHigh` trait can be implemented using an implementation of the `ParseLow` trait. This is accomplished with a type called the `GenericVisitor`, whose implementation closely resembles the algorithm in figure 3.

The `GenericVisitor` is meant to be called by a *real* visitor for a framework's testing language. For example, support for Rust's testing framework includes an implementation of `syn`'s `Visit` trait.[7] That `Visit` implementation calls various `GenericVisitor` methods while visiting a `syn` syntax tree. The `GenericVisitor`, in turn, interacts with the framework through the `ParseLow` trait interface.

[7]Recall from table I that Necessist uses `syn` to parse Rust.

```
impl<..., 'ast, T: ParseLow> GenericVisitor<..., 'ast, T> {
  ...
  pub fn visit_statement(
    &mut self, ...,
    statement: <T::Types as AbstractTypes>::Statement<'ast>,
  ) -> bool { ...}
  pub fn visit_statement_post(
    &mut self, ...,
    statement: <T::Types as AbstractTypes>::Statement<'ast>,
  ) { ... }
  ...
}
```

Fig. 5. Excerpt of the `GenericVisitor`'s implementation

```
pub trait ParseLow: Sized {
  type Types: AbstractTypes;
  const IGNORED_FUNCTIONS: Option<&'static [&'static str]>;
  ...
  fn statement_is_control<'ast>(
    &self, ...,
    statement: <Self::Types as AbstractTypes>::Statement<
      'ast,
    >,
  ) -> bool;
  ...
}
```

Fig. 6. Excerpt of the `ParseLow` trait's definition

Excerpts of the `GenericVisitor` and `ParseLow` trait appear in figures 5 and 6 (respectively).[8,9] The `visit_statement` and `visit_statement_post` methods are essentially the callbacks from figure 3. The `GenericVisitor` currently has eight such callbacks: `visit_test`, `visit_statement`, `visit_call`, and `visit_macro_call`, along with their corresponding `post` methods.

Note that the `ParseLow` trait requires its implementations to specify a type `Types`, which implements a trait `AbstractTypes` (not pictured here). The trait `AbstractTypes` includes several generic associated types (GATs) [55], [56]. The GATs abstract away the details of the framework's syntax tree from the `GenericVisitor`. For example, `Statement` is the concrete type of a framework's syntax tree statements. The types are generic in that they are parameterized by the lifetime of the syntax tree (`'ast`).

The `ParseLow` trait includes methods statement_is_control (pictured) and statement_is_declaration (not pictured). These methods correspond to the functions is_control and is_declaration in figure 3. The `ParseLow` trait currently consists of 18 methods in total. Most of the additional methods exist to provide information to the `GenericVisitor`, e.g., statement_is_expression, expression_is_call, etc.

Finally, note the array `IGNORED_FUNCTIONS` within the `ParseLow` trait. Not pictured are two other similar arrays, `IGNORED_MACROS` and `IGNORED_METHODS`. These three arrays contains the functions, macros, and methods (respectively) that are ignored by default for a framework. For example, for the Anchor and Hardhat frameworks, the function `assert` is ignored by default. Necessist allows users to extend these arrays via a configuration file. The extended arrays are used by the `GenericVisitor` to implement the is_ignored_call function in figure 3.

[8]https://github.com/trailofbits/necessist/blob/1ded71dc3bbd191535d38e9b6c1467eda7ea42b2/frameworks/src/generic_visitor.rs#L82-L458

[9]https://github.com/trailofbits/necessist/blob/1ded71dc3bbd191535d38e9b6c1467eda7ea42b2/frameworks/src/parsing.rs#L74-L171

B. Execution

Compared to parsing, executing each framework's test as described in section II-B is relatively straightforward. Like parsing, execution is accomplished with two traits, `RunHigh` and `RunLow`. The `RunHigh` trait contains just two methods, `dry_run` and `exec`, which correspond to the two uses of `run` on lines 4 and 16 of figure 4. The `RunLow` trait contains three methods, which return commands to run an entire test file, build a test file, or run an individual test.

The Foundry, Go, and Rust frameworks implement the `RunHigh` trait with an implementation of the `RunLow` trait. In contrast, the Anchor and Hardhat frameworks implement `RunHigh` directly. The reason for this is that `RunLow` is designed to run tests individually. However, Anchor and Hardhat are built on top of Mocha [57], which, unlike other frameworks, allows tests to be interdependent. More specifically, Mocha guarantees the order in which certain operations are performed. As a result, many real-world Anchor and Hardhat tests expect certain operations to have been performed by previous tests. Thus, attempting to run Anchor and Hardhat tests individually would result in many unwarranted failures.

C. Supporting additional frameworks

Adding support to Necessist for a new testing framework requires essentially three things: a Rust implementation of a parser for the framework's testing language, an implementations of the `ParseHigh` trait, and an implementations of the `RunHigh` trait. If the parser provides a visitor interface, then implementing the `ParseHigh` trait is relatively straightforward using the `GenericVisitor` and an implementation of the `ParseLow` trait.

IV. Experimental Results

In this section, we consider the following questions. How does Necessist's runtime on a test file compare to its number of removal candidates? How does a test file's number of removal candidates compare to its number of lines? And, finally, could past bugs have been found by running Necessist?

To answer these questions, we run Necessist on 400 files drawn from past commits to the Go standard library. The results show that while Necessist is an expensive tool to run, its runtime is reasonable. Furthermore, Necessist is able to find past bugs in the Go standard library, providing evidence that Necessist can find future bugs.

A. Data set

Our data set consisted of 400 files drawn from past commits to the Go standard library. We chose the Go standard library for the following reasons. First, of the libraries to which Necessist is currently applicable, the Go standard library is the oldest. It has thus been used by, and received contributions from, a wide range of developers. Second, the Go standard

library is tested with standard Go tools (e.g., `go test`). This makes it easy to test with Necessist. Compare this to, say, the Rust standard library, which uses a custom script (`x.py`) for testing.

We searched the Go repository from tag `go1.21.1` back through commit `781da44` (2019-03-12) for fixes to tests.[10] Concretely, we searched for commits with log messages matching the following `grep` regular expression:

```
\<fix\>[^.]*\(\<test\|_test\)
```

Our search resulted in 140 commits; however, one could not be built.[11] The 139 remaining commits involved changes to 144 different test files, though many were changed by multiple commits.[12] Counting with multiplicity, exactly 200 test files were changed.

B. Experimental setup

For each of the 200 files, we performed two runs of Necessist. One run was performed on the file in its state just prior to the commit; a second run was performed on the file in its state just after the commit. This resulted in a total of 400 runs. For each run, we recorded Necessist's output as well as its runtime. We collected this data on a a Digital Ocean instance with 96 GB of RAM and 48 CPUs. We ran six instances of a debug build of Necessist concurrently. Each instance was restricted to a distinct set of eight CPUs.

C. Results

Figure 7 shows the number of removal candidates relative to the number of lines in the test file, for each of the 400 runs. Figure 8 shows Necessist's runtime relative to the number of removal candidates, for each of the 400 runs. Note that, for both graphs, both axes are in log scale. Also note that the graphs distinguish the runs prior to a commit (squares) from the runs after (diamonds).

As figure 7 shows, the number of removal candidates increases roughly linearly relative to the number of lines in the test file. Similarly, figure 8 shows that runtime increases roughly linearly relative to the number of removal candidates.

The file with the largest number of removal candidates was `cmd/go/go_test.go` as changed by commit `746f405` (2019-03-17). This file contained 6250 lines prior to the commit, and 6252 after. Necessist identified 4280 removal candidates in the file prior to the commit, and 4282 after.

The file that caused Necessist to run for longest was `net/http/transport_test.go` at commit `d05c035` (2020-08-06). This file had 1654 removal candidates both before and after the commit. Necessist took 10029 seconds (∼2.8 hours) in the run prior to the commit, and 10036 seconds after.

[10]Commit `781da44` is when Go switched to using modules within the Go repository. Building commits from prior to then with modern tools is more difficult.

[11]Attempting to build commit `430cee7` (2020-05-05) produced the error message: `invalid GOAMD64: must be v1, v2, v3, v4`. We were unable to alleviate the error.

[12]The file `net/http/transport_test.go` was changed by the most commits (seven).

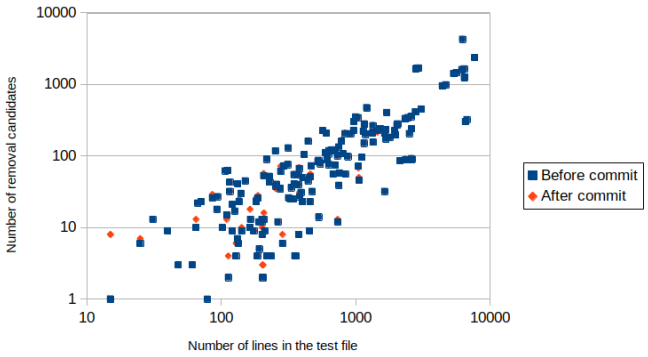

Fig. 7. The number of removal candidates increases roughly linearly relative to the number of lines in the test file.

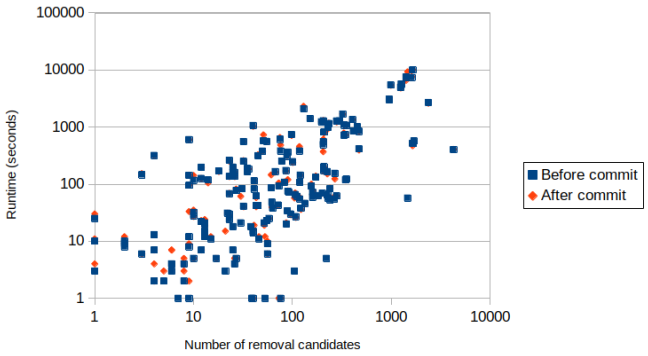

Fig. 8. Necessist's runtime increases roughly linearly relative to the number of removal candidates.

D. Past bugs found

Could past bugs in the Go standard library have been found by running Necessist? Performing an exhaustive review of all 139 commits would be impractical. However, if one looks for cases where "passed" outcomes were removed by a commit, then several examples pop out quickly. We highlight three.

The first two examples are similar in spirit to figure 1. Figure 9 gives an excerpt of the test fixed by commit `979d65d` (2019-11-04).[13] The test is meant to verify that a call to a function `SendMail` succeeds. The test uses a Go routine to act as a mock mail server, and a `WaitGroup` [58] to wait for the routine to complete. However, if the statement `wg.Add(1)` is removed, then the call to `wg.Wait()` returns immediately, yet the test still passes. Thus, if an error occurs in the Go routine (e.g., the call to `Accept` fails), the test still passes. The fix introduced by `979d65d` (not pictured here) is to use an error channel instead of a `WaitGroup`.[14] In particular, the test now waits for a possibly `nil` error to

[13]https://github.com/golang/go/blob/9a0a82445650eebedf5633fdfe6e73b5836dc5c9/src/net/smtp/smtp_test.go#L639-L681

[14]https://github.com/golang/go/blob/979d65dbc7687ca4c0bc76576d37affae7b7c041/src/net/smtp/smtp_test.go#L646-L683

```
func TestSendMailWithAuth(t *testing.T) {
  ... wg := sync.WaitGroup{} ...
  go func() {
    defer wg.Done()
    conn, err := l.Accept()
    if err != nil {
      t.Errorf("Accept error: %v", err)
      return
    }
    ...
  }()
  wg.Add(1)
  err = SendMail(..., []byte(strings.Replace(
`From: test@example.com
...
`, "\n", "\r\n", -1)))
  ...
  wg.Wait()
}
```

Fig. 9. Past bug in `net/smtp/smtp_test.go` found by Necessist. The test passes with the statement `wg.Add(1)` removed. Thus, if the call to `Accept` fails, the test passes.

```
ts.CloseClientConnections() // surprise!
// This test has an expected race. Sleeping for 25 ms
// prevents it on most fast machines, ...
time.Sleep(25 * time.Millisecond)
```

Fig. 10. Slightly reformatted version of the code removed by commit `3970667`. Necessist reports that the call to `Sleep` is unnecessary, highlighting that there is a race condition.

be sent on the channel. The test fails if a non-`nil` error is received on the channel.

The second example (not pictured here, but again similar figure 1) concerns a fix introduced by commit `42f8199` (2020-02-13) to a test `TestMutexStress`.[15] This test spawns several Go routines, and each performs a set of random operations on a structure called an `FDMutex`. The test waits for each of the Go routines to send an empty struct on a `done` channel before the test completes. However, if an error occurs in one of the Go routines, the test hangs. Necessist reveals that there is a problem by showing that the test passes even when no receives are performed on the `done` channel.

As a third example, Necessist rediscovers a known race condition in a test in `net/http/transport_test.go`. The code in Figure 10 was removed by commit `3970667` (2019-05-25).[16] When run on the file prior to the commit, Necessist reports that the call `Sleep` is unnecessary, highlighting that there is a race condition.

V. Discussion

Based on our experience using Necessist, its main limitations are its runtime and its tendency to produce false positives. This section discusses those limitations, and ways for dealing with them.

A. Runtime

Removing a statement or method call from a test requires that the test be rebuilt. Thus, much of Necessist's runtime is spent recompiling tests. As mentioned in section I-D, running Necessist on a moderately sized codebase (say, a few tens of KLOC), can take several hours.

Ideally, only the test that has been modified would need to be rebuilt. In other words, the compilation results produced for the other tests could be reused. In other settings, this idea is called incremental compilation [59]. Incremental compilers exist for some languages that Necessist supports (e.g., Rust [60]). But, to our knowledge, no such compiler performs incremental compilation at the granularity of tests. Adding to this, Necessist currently operates only on syntax. Accounting for compilation artifacts would require a considerable redesign of Necessist. Moreover, it would likely require integrating Necessist with a different compiler for each testing language.

[15]https://github.com/golang/go/blob/ea3bfba87cfd7141870f975102029e2e341b4af3/src/internal/poll/fd_mutex_test.go#L146-L218

[16]https://github.com/golang/go/blob/f736de04aa52d4889760ecfe4380da01aaf4758f/src/net/http/transport_test.go#L776-L782

```
#[test]
fn test_sort() {
  let xs = (0..10).collect::<Vec<_>>();
  let mut ys = xs.clone();
  ys.shuffle();
  ys.sort();
  assert_eq!(xs, ys);
}
```

Fig. 11. Removing the statement `ys.shuffle();` weakens the test, and thus produces a false positive.

B. False positives

For Necessist, a false positive is the removal of a statement or method call that results in a passing test, but that does not reflect a bug. Put another way, it is a statement or method call that Necessist incorrectly reports as unnecessary.

Generally speaking, most of Necessist's false positives can be put into one of three categories, based on the purpose of the statement or method call removed. The three categories are: *test setup*, *calls that perform checks*, and *subordinate tests*. This subsection discusses the three categories, and ways for dealing with them.

Test setup. By this, we mean a statement or method call that modifies the environment in which the test is run. Removing the statement results in either a weaker test or one that is conceptually distinct. Figure 11 gives an example of a pattern that occurs frequently in tests. This example verifies that a call to a method `sort` undoes the effects of a call to another method `shuffle`. If one were to run Necessist on this example, it would report that removing the statement `ys.shuffle();` results in a passing test. But, clearly, the statement is necessary for the test to be meaningful.

One way of dealing with this class of false positives is to add assertions to verify that the setup was successful. In the case of figure 11, for example, one could add a statement such as the following just after the call to `shuffle`:

```
assert_ne!(xs, ys);
```

Note that adding this assertion makes the call to `shuffle` necessary for the test to pass, because removing the call would cause the assertion to fail. Put another way, adding the assertion increases confidence that `sort` is doing meaningful

```
#[test]
fn test_allocator() {
  let mut allocator = Allocator::new();
  for _ in 0..N {
    perform_random_operation(&mut allocator);
    allocator.check_invariants();
  }
}
```

Fig. 12. If method call `.check_invariants()` performs checks, removing it produces a false positive.

work, and is not simply a no-op. This last point is worth emphasizing. Even though these false positives are, strictly speaking, faults of Necessist, we find that addressing them in source code often results in better tests.

Calls that perform checks. Often, some set of checks is needed to be performed by multiple tests. In line with good development practices, developers often group such checks together in a function. However, such functions cause difficulty for Necessist. Not realizing their purpose, Necessist removes calls to these functions and reports that the tests containing them pass.

Figure 12 gives a hypothetical example. The test repeatedly performs a randomly chosen operation on an `Allocator` (i.e., memory allocator), followed by a call to a method `check_invariants`. One might imagine that `check_invariants` verifies that the sum of all allocations is equal to the total size of all currently allocated and freed memory, for example. If one were to run Necessist on this example, it would report that removing the method call `.check_invariants()` results in a passing test. But, clearly, the method call is necessary for the test to be meaningful.

The ability to extend the sets of ignored functions, macros, and methods (mentioned in section III-B) was added specifically to deal with such cases. For the case of figure 12, one could specify that the method `check_invariants` should be ignored. Doing so would prevent Necessist from removing calls to that method, and thereby eliminate the associated reports of passing tests.

Subordinate tests. Often, a "test" as understood by the testing framework, is actually a collection of many small "tests," conceptually. One way this arises is when a developer has a set of *test cases* (e.g., inputs paired with the outputs they should generate), and writing a separate test for each test case would result in many, nearly identical tests. To avoid duplicating code, the developer instead handles all of the test cases within a single test.

A hypothetical example appears in figure 13. If one were to run Necessist on this example, it would report that removing the statement `run_test(test_case);` results in a passing test. But, clearly, the statement is necessary for the test to be meaningful.

Necessist does not currently have a good solution for this problem, though a relevant enhancement is planned [61]. Specifically, the function that handles test cases (e.g., `run_test` in figure 13) is often defined within the same file as the test that calls it. In such cases, Necessist could visit

```
#[test]
fn test_many() {
  for test_case in TEST_CASES { run_test(test_case); }
}
```

Fig. 13. If statement `run_test(test_case);` runs subordinate tests, removing it produces a false positive.

the function body and select statements and method calls for removal, just as it does for tests.

VI. Future work and conclusion

We see at least two directions in which research on Necessist could proceed.

Explore other types of mutations. As mentioned in section I-D, conventional mutation testing tools perform many more types of mutations than Necessist does. It might make sense for Necessist to support additional mutations. For example, Necessist could change some numeric literals in a way that is consistent with its current behavior. To see this, note that `x += 2;` is equivalent to `x += 1; x += 1;`. Thus, if Necessist were to change the `2` to a `1`, it would be essentially the same as rewriting `x += 2;` as `x += 1; x += 1;` and then removing one of the latter two statements.

Incorporate semantic information. Necessist currently does not consider type or name resolution information (see section II-A). Using such information could allow Necessist to produce more precise results, e.g., by better distinguishing the functions, macros, and methods it should ignore. Incorporating such information in a framework agnostic way could require some clever software engineering.

In summary, Necessist is tool for finding certain types of bugs in tests. We described the method that it uses abstractly (section II), and described an implementation that works on real-world code that uses Anchor, Foundry, Go, Hardhat, or Rust (section III). We also showed that Necessist is able to find past bugs in the Go standard library, providing evidence that Necessist can find future bugs (section IV).

References

[1] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, 2nd ed. Wiley Publishing, 2018.

[2] H. Ma, "A survey of modern compiler fuzzing," 2023. [Online]. Available: https://arxiv.org/abs/2306.06884

[3] A. Arya. Short intro to OSS-Fuzz. [Online]. Available: https://www.code-intelligence.com/blog/intro-to-oss-fuzz

[4] B. Vuleta. Software testing statistics - 2023. [Online]. Available: https://truelist.co/blog/software-testing-statistics/

[5] The Go Programming Language. Go. [Online]. Available: https://github.com/golang/go

[6] The Rust Programming Language. Rust. [Online]. Available: https://github.com/rust-lang/rust

[7] Kitware. (2023) Testing with CMake and CTest. [Online]. Available: https://cmake.org/cmake/help/book/mastering-cmake/chapter/Testing%20With%20CMake%20and%20CTest.html

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, jun 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[9] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 328–343. [Online]. Available: https://doi.org/10.1145/3064176.3064183

[10] U. Kirstein. Post-mortem analysis of the Notional Finance vulnerability — a tautological invariant. [Online]. Available: https://medium.com/certora/post-mortem-analysis-of-the-notional-finance-vulnerability-a-tautological-invariant-574d02d6ac15

[11] Trail of Bits. Necessist. [Online]. Available: https://github.com/trailofbits/necessist

[12] S. Fackler. rust-openssl. [Online]. Available: https://github.com/sfackler/rust-openssl

[13] OpenSSL. (2023) Cryptography and SSL/TLS toolkit. [Online]. Available: https://www.openssl.org/

[14] coral-xyz. Anchor. [Online]. Available: https://github.com/coral-xyz/anchor

[15] Coral. WTF are xNFTs? [Online]. Available: https://www.coral.community/post/wtf-are-xnfts

[16] coral-xyz. xNFT. [Online]. Available: https://github.com/coral-xyz/xnft

[17] SWC. SWC. [Online]. Available: https://github.com/swc-project/swc

[18] Paradigm. Foundry. [Online]. Available: https://github.com/foundry-rs/foundry

[19] G. Konstantopoulos. Introducing the Foundry Ethereum development toolbox. [Online]. Available: https://www.paradigm.xyz/2021/12/introducing-the-foundry-ethereum-development-toolbox

[20] Hyperledger Foundation. Solang. [Online]. Available: https://github.com/hyperledger/solang

[21] tree-sitter. tree-sitter. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[22] Nomic Foundation. Hadhat. [Online]. Available: https://github.com/NomicFoundation/hardhat

[23] The Rust Team. (2018-12-06) Rust. [Online]. Available: https://www.rust-lang.org/

[24] D. Tolnay. syn. [Online]. Available: https://github.com/dtolnay/syn

[25] T. Fidry. Awesome mutation testing. [Online]. Available: https://github.com/theofidry/awesome-mutation-testing

[26] B. Wang. Accmut. [Online]. Available: https://github.com/wangbo15/accmut

[27] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 402–411. [Online]. Available: https://doi.org/10.1145/1062455.1062530

[28] J. Brännström. Dextool mutate. [Online]. Available: https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate

[29] Mull Project. Mull. [Online]. Available: https://github.com/mull-project/mull

[30] M. Kim. MUSIC. [Online]. Available: https://github.com/swtv-kaist/MUSIC

[31] N. Lohmann. Mutate++. [Online]. Available: https://github.com/nlohmann/mutate_cpp

[32] F. Hariri and A. Shi, "SRCIROR: A toolset for mutation testing of C source code and LLVM intermediate representation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 860–863. [Online]. Available: https://doi.org/10.1145/3238147.3240482

[33] Avito Tech. go-mutesting. [Online]. Available: https://github.com/avito-tech/go-mutesting

[34] Gremlins. Gremlins. [Online]. Available: https://github.com/go-gremlins/gremlins

[35] G. J. Tramontina. Ooze. [Online]. Available: https://github.com/gtramontina/ooze

[36] M. Pool. cargo-mutants. [Online]. Available: https://github.com/sourcefrog/cargo-mutants

[37] A. Bogus. mutagen. [Online]. Available: https://github.com/llogiq/mutagen

[38] D. Xu. Deviant. [Online]. Available: https://github.com/dianxiangxu/Deviant

[39] Certora. Gambit. [Online]. Available: https://github.com/Certora/gambit

[40] C. Lee. MuSC tool demo. [Online]. Available: https://github.com/belikout/MuSC-Tool-Demo-repo

[41] Y. Ivanova and A. Khritankov, "RegularMutator: A mutation testing tool for Solidity smart contracts," *Procedia Computer Science*, vol. 178, pp. 75–83, 2020, 9th International Young Scientists Conference in Computational Science, YSC2020, 05-12 September 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920323826

[42] M. Barboni. SuMo. [Online]. Available: https://github.com/MorenaBarboni/SuMo-SOlidity-MUtator

[43] J. Honig. Vertigo. [Online]. Available: https://github.com/JoranHonig/vertigo

[44] A. Groce, I. Ahmed, C. Jensen, P. E. Mckenney, and J. Holmes, "How verified (or tested) is my code? Falsification-driven verification and testing," *Automated Software Engg.*, vol. 25, no. 4, p. 917–960, dec 2018.

[45] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 25–28. [Online]. Available: https://doi.org/10.1145/3183440.3183485

[46] U. S. Shah and D. C. Jinwala, "Resolving ambiguities in natural language software requirements: A comprehensive survey," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 5, p. 1–7, sep 2015. [Online]. Available: https://doi.org/10.1145/2815021.2815032

[47] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. V. Nguyen, N. Aguirre, and M. Frias, "Artifact of bounded exhaustive search of Alloy specification repairs," in *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '21. IEEE Press, 2021, p. 209–210. [Online]. Available: https://doi.org/10.1109/ICSE-Companion52605.2021.00093

[48] G. Zheng, T. V. Nguyen, S. G. Brida, G. Regis, N. Aguirre, M. F. Frias, and H. Bagheri, "Atr: Template-based repair for Alloy specifications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 666–677. [Online]. Available: https://doi.org/10.1145/3533767.3534369

[49] S. Gutiérrez Brida, G. Regis, G. Zheng, H. Bagheri, T. V. Nguyen, N. Aguirre, and M. Frias, "ICEBAR: Feedback-driven iterative repair of Alloy specifications," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556944

[50] Y. Megdiche, F. Huch, and L. Stevens, "A linter for Isabelle: Implementation and evaluation," 2022. [Online]. Available: https://arxiv.org/abs/2207.10424

[51] Lean Community. (2023) Default linters. [Online]. Available: https://leanprover-community.github.io/mathlib_docs/tactic/lint/default.html

[52] F. Yergeau, G. Adams, M. J. Dürst, and G. T. Nicol, "Internationalization of the Hypertext Markup Language," RFC 2070, Jan. 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2070

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[54] The Rust Standard Library. (2023) Struct `std::process::Command`. [Online]. Available: https://doc.rust-lang.org/std/process/struct.Command.html

[55] S. Shipwreckt. Tracking issue for generic associated types (GAT). GitHub issue #44265. [Online]. Available: https://github.com/rust-lang/rust/issues/44265

[56] J. Huey. Generic associated types to be stable in Rust 1.65. [Online]. Available: https://blog.rust-lang.org/2022/10/28/gats-stabilization.html

[57] OpenJS Foundation. (2023) Mocha. [Online]. Available: https://mochajs.org/

[58] The Go Programming Language. (2023) type WaitGroup. [Online]. Available: https://pkg.go.dev/sync#WaitGroup

[59] J. L. Ryan, R. L. Crandall, and M. C. Medwedeff, "A conversational system for incremental compilation and execution in a time-sharing environment," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 1–21. [Online]. Available: https://doi.org/10.1145/1464291.1464293

[60] M. Woerister. Incremental compilation. [Online]. Available: https://blog.rust-lang.org/2016/09/08/incremental.html

[61] S. Moelius. Response to "removal of closures and functions that contain assert macros cause false positives". GitHub issue #502. [Online]. Available: https://github.com/trailofbits/necessist/issues/502#issuecomment-1592825871