Dynamic Human-in-the-Loop Assertion Generation

Lucas Zamprogno, Braxton Hall, Reid Holmes, *Member, IEEE,* and Joanne M Atlee, *Member, IEEE*

Abstract—

Test cases use assertions to check program behaviour. While these assertions may not be complex, they are themselves code that must be written correctly in order to determine whether a test case should pass or fail. We claim that most test assertions are relatively repetitive and straight-forward, making their construction well suited to automation and that this automation can reduce developer effort while improving assertion quality. Examining 33,873 assertions from 105 projects revealed that developer-written assertions fall into twelve high-level categories, confirming that the vast majority (>90%) of test assertions are fairly simple in practice. We created AutoAssert, a human-in-the-loop tool to fit naturally into a developer's test-writing workflow by automatically generating assertions for JavaScript and TypeScript test cases. A developer invokes AutoAssert by identifying the variable they want validated; AutoAssert uses dynamic analysis to generate assertions relevant for this variable and its runtime values, injecting the assertions into the test case for the developer to accept, modify, delete. Comparing AutoAssert's assertions to those written by developers, we found that the assertions generated by AutoAssert are the same kind of assertion as was written by developers 84% of the time in a sample of over 1,000 assertions. Additionally we validated the utility of AutoAssert-generated assertions with 17 developers who found the majority of generated assertions to be useful and expressed considerable interest in using such a tool for their own projects.

1 Introduction

Automated testing continues to grow in importance for modern software development. There are many kinds of automated tests: unit tests can provide quick feedback for developers [1], integration tests can ensure components work in concert [2], and smoke tests provide rapid-high level feedback [3]; taken together these tests all play a central role in regression testing [4]. One commonality among these kinds of automated testing is that each test case needs assertions to verify the behaviour of the code under test.

A test case comprises code that invokes a behaviour in the code under test and a set of assertions that validate if that behaviour is as expected. Assertions play an out-sized role in test cases [5] because they serve as oracles: a test case passes if all of its assertions pass; the test case fails if any of its assertions fail. Despite the fact that assertions themselves are short and somewhat repetitive segments of code, prior work has shown that support is needed to reduce the cost of identifying test oracles [6] and that developer-written test case assertions need to be strengthened [7].

Consequently, several researchers have investigated how to generate assertions automatically. Assertion generation approaches are broadly classified as being either static or dynamic. Static approaches (e.g., UnitPlus [8], Obsidian [9], Atlas [10]) have the benefit of being fast; but developers need to verify that the asserted values make sense for their system and that the generated assertions actually pass, as the generated values used in assertions are not based on actual values that the variables will hold at runtime. Dynamic approaches tend to be slower as they must execute the code under test to inspect runtime values in order to generate assertions will pass when they are executed.

Existing dynamic approaches (e.g., Eclat [11], ZoomIn [12]) lack a way of knowing which aspects of the code under test the tester wants to validate. Without this information, these approaches create assertions for *any* variable in the test file and after *every* time a variable is modified. The resulting test cases do not resemble those written by developers: the test cases are bloated with assertions and this impairs the tests' ability to serve as documentation for expected behaviour. Unfortunately, assertion generation approaches have been broadly evaluated in terms of some form of precision, but have thus far not been evaluated with developers.

In this paper we present how a human-in-the-loop approach can address shortcomings of dynamic approaches to assertion generation. Specifically, we investigate how developers write assertions and introduce and evaluate a tool, called AutoAssert, that developers can use to interactively generate assertions for their test cases as they write them. AutoAssert is designed to fit into a developer's normal test-writing workflow. As the developer is creating or modifying a test case, they select a variable in their test case for which they want AutoAssert to quickly generate assertions. AutoAssert then generates and inserts one to four passing assertions into the test case that the developer examines to keep, delete, or modify as they see fit. AutoAssert uses dynamic analysis to observe the runtime value of a selected variable and determines the most appropriate assertions for the variable. The human-in-the-loop aspect of the tool is important: the developer selects the variables they want assertions for, the tool generates assertions only for these variables using only valid values, and the developer evaluates the relatively small number of generated assertions, thereby ensuring that only assertions they think are useful are persisted in their test case. AutoAssert accounts for runtime variations (e.g., non-determinism) and can customize generated assertions to match a project's style.

- *Lucas Zamprogno, Braxton Hall, and Reid Holmes are with the Department of Computer Science, University of British Columbia, Vancouver, Canada.*
  E-mail: lucasaz@cs.ubc.ca, braxtonh@cs.ubc.ca, rtholmes@cs.ubc.ca
- *Joanne M Atlee is with the Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada.*
  Email: jmatlee@uwaterloo.ca

Manuscript submitted October 27, 2021; accepted October 22, 2022.

More specifically, the primary goal of this research is to investigate the feasibility of generating test assertions with a human-in-the-loop system and evaluate the utility of these assertions for industrial developers. To do this, we employed the following methodology: We first performed a quantitative analysis of developer-written assertions to understand what these look like in practice (Section 2). Using this understanding, we built a proof-of-concept tool called AutoAssert to automatically generate the most commonly occurring assertions (Section 3). Using a quantitative simulation study we compared the assertions generated by AutoAssert to those written by developers, and show that AutoAssert can generate the same kind of assertions as developers 84% of the time (Section 4). Finally, we evaluated the utility of the AutoAssert tool in a formative evaluation with real developers, which is uncommon for assertion generation approaches (Section 5).

This paper makes the following contributions:

- An empirical study characterizing the size, complexity, and semantics of developer-written assertions.
- A prototype tool called AutoAssert that supports human-in-the-loop assertion generation.
- An empirical simulation comparing the assertions generated by AutoAssert to those originally written by developers.
- A formative user study examining how developers perceived the utility of automatically generated assertions for real test cases.

This work characterizes developer-written assertions and demonstrates a precise approach that can automatically generate the majority of these assertions. We also extend prior work by showing that developers are able to interpret automatically generated assertions and find them useful while working with test cases.

2 Assertions in Practice

We first sought to understand how developers write assertions in practice by investigating:

RQ1: What do test assertions look like in practice?

This research question examined the scope, complexity, and semantic variation of developer-written assertions in real software systems. The insight from this investigation guided the design of our assertion-generation approach to ensure it can create the kinds of assertions that developers actually write. By characterizing real-world developer-written assertions, this study can also help others better understand what these assertions look like in practice.

2.1 Methodology

To answer **RQ1**, we performed a quantitative study by statically analyzing developer-written test assertions present in open-source projects. We selected these projects by examining all JavaScript and TypeScript projects published on npm that are dependent on the Chai assertion library and Mocha test framework.

Mocha and Chai are common JavaScript unit testing frameworks, analogous to JUnit in Java. From these projects, we selected projects with at least 100 stars to try to capture maintained systems and filter out personal projects. This resulted in 105 open-source projects; we analyzed all 33,873 test assertions from 18,937 test cases present in these.

We used the TypeScript parser (which can also parse JavaScript) to create a static analysis tool to analyze each of these assertions. The analysis tool first checked whether an assertion was non-trivial, that is whether it actually asserted on a variable, in contrast to a trivial assertion like `expect.fail()`. This reduced the number of assertions from 33,873 to 33,650. For the 33,650 non-trivial assertions, the analysis tool extracted:

1) *Element under test.* Elements under test were property accesses (35.8%), variables holding the result of method calls (31.3%), or inline method calls (18.1%).
2) *Assertion method.* The assertion library API declares the available assertion methods (e.g., `equals`, `include`). For this study we included all methods declared by the Chai assertion library.
3) *Expected value.* The element under test is compared against an expected value for correctness. This was most commonly a literal value such as a string or a number (61.6%). Chai methods were often used as expected values, for example `expect(val).to.be.true` (24%). For the remaining assertions, the expected values were identifiers (12.3%), property accesses (6%), complex expressions (3.4%), or method calls (2.7%). As a single assertion can involve more than one expected value, percentages add to more than 100%.

2.2 Results

We next analyzed the assertions to learn how many were used in each test case, how complex they were, what kinds of assertions developers use, and how they were constructed.

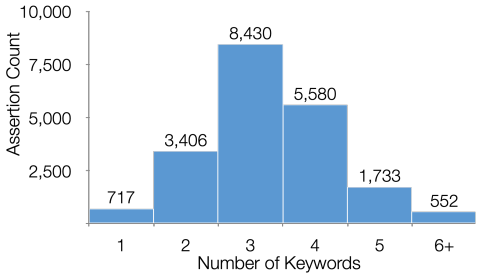

Fig. 1. Histogram showing the number of keywords per assertion across assertions using the `expect` API.

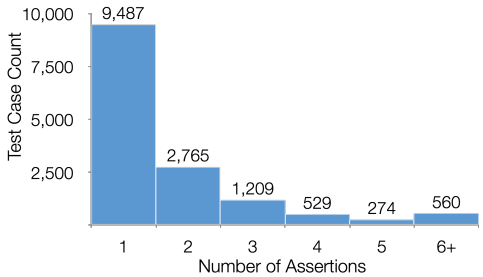

Fig. 2. Histogram showing the number of assertions per test case across 14,824 test cases containing at least one inline assertion.

TABLE 1
Assertions encountered in practice. This shows the twelve high-level assertion categories categorized from 33,650 non-trivial assertions contained in the 18,937 tests from the 105 analyzed projects. Some assertions can appear in multiple categories resulting in an overall percentage greater than 100. Categories are based on assertion semantics and have been adjusted to account for semantic equivalents as described in Section 2.2.4.

Category	%	Count	Description	Representative assertion operators
Equality	39.3%	13,325	Exact matches of values or references	`to.equal`, `to.eq`
Boolean	14.3%	4,854	Value is `true` or `false`	`to.be.true`, `to.be.false`
Inclusion	7.1%	2,409	Element present in arrays or strings	`to.include`, `to.have.members`
Length	6.7%	2,259	Array or string length	`to.have.length`, `to.not.be.empty`
Existence	6.1%	2,073	Value is `null` or `undefined`	`to.exist`, `to.be.null`
Properties	4.8%	1,610	Existence and/or values of object properties	`to.have.keys`, `to.have.property`
Calls	4.0%	1,369	Method has been called, check arguments	`to.be.called`, `to.be.calledOnce`
Type	3.3%	1,101	Primitive types and class instances	`to.be.a`, `to.be.instanceOf`
Numeric	2.2%	736	Comparisons of numeric values to each other	`to.be.below`, `to.be.at.least`
Throw	2.0%	659	Function throws or returns successfully	`to.throw`, `to.not.throw`
Patterns	1.9%	645	Regular expressions and pattern matching	`to.match`, `to.have.matches`
Truthiness	1.8%	619	Value can be coerced to `true` or `false`	`to.be.ok`, `to.be.falsy`
Uncategorized	6.3%	2,147	Does not fit other categories, specific plugins	`to.be.fulfilled`, `to.have.style`
Invalid	2.1%	717	API misuse	`[No assertion operator]`

2.2.1 Assertion Density

Developers add assertions to their test cases to validate program behaviours; in practice we find that most tests contain few assertions. Of the 18,937 test cases, 4,113 test cases (22%) contain no test assertions[1] or rely on helper functions to assert behaviour. Our dataset omits tests containing no assertions to avoid skewing the mean; we also omitted tests that invoke assertions from helper functions.

Figure 2 shows the distribution of the number of assertions per test case among the remaining 14,824 tests that contain at least one assertion: these test cases have a mean of 1.4 assertions per test and a median of 1 assertion per test; 17.4% of the test cases have three or more assertions; and the outlying test case with the most assertions contained 183 assertions. These results argue against writing assertions for every variable and after every change in variable value.

> □ Developers typically include a low number of assertions per test case; assertion generation techniques should focus on producing a small set of key assertions.

2.2.2 Assertion Complexity

In practice, most assertion statements are relatively straightforward. To examine the complexity of assertions, we counted the number of keywords from the assertion library present in each assertion statement. Chai's `expect` API allows for building complex assertion statements by chaining these keywords, so this count can serve as a proxy for an assertion's complexity. Some examples of assertion statements and their **keyword** counts are:

#	Assertion Statement
2	`expect(val).throws(err);`
3	`expect(journal).to.equal('TSE');`
4	`expect(reviewers).to.not.include('r2');`
11	`expect(pages).to.exist.and.to.be.at.least(10).and.at.most(13);`

1. Some assertion-free tests simply check to see if the code under test throws an error, which implicitly causes the test case to fail.

Figure 1 shows the assertion complexity across the 20,418 assertions using the `expect` assertion form. Although Chai supports two forms of assertions (written with either the `expect` or `assert` keyword) projects overwhelmingly use one form or the other. The two forms are semantically identical and the number of assertions in either form would be the same, but assertions written using `assert` are more compact and use fewer keywords than assertions written using `expect`. To increase the internal consistency of our assertion-complexity analysis, we examined only `expect` assertions because that form is more commonly used and is more verbose, leading to results of higher complexity. This only removed 12% of the developer-written assertions from the analysis, as 88% of examined assertions used the `expect` syntax. Of the 20,418 `expect` assertions analyzed, most contain three or four keywords, which typically corresponds to a single check (three keywords) or the negation of a check (four keywords) in the Chai assertion library. While the last example above shows that it is possible to write more complex compound assertions in Chai, it turns out that developers do not seem to do this often in practice.

> □ Most developer-written assertions are simple; assertion generation techniques should favour simple assertions over complex assertions.

2.2.3 Assertion Categories

To understand the kinds of assertions developers authored, we categorized all 33,650 non-trivial assertions in the dataset. We grouped assertions by their keywords, discarding those keywords that were used only as modifiers (e.g., `not`, `deep`) or syntactic sugar (e.g., `to`, `have`); this left only the keywords representing specific assertion semantics. To understand the semantics of each keyword, we used the keyword name and the official documentation from the assertion library to confirm the kind of behaviours validated by each keyword. We then performed an open card sort [13] on the per-keyword list to identify categories of semantic behaviours validated by developers in practice. For example, the keywords `length`, `size`, and `empty` would be grouped into the same category LENGTH. While these keywords

are each independent, they all evaluate the same kind of semantic property for the value under test.

The result is twelve categories of assertions that developers commonly expressed in their test cases to validate the behavioural correctness of the code under test. Table 1 lists the categories and percentage of assertions classified in each category. The results indicate that developers validate a breadth of behaviours and that most of these behaviours (>90%) fall into a reasonable number of categories.

> □ Developers create assertions to check a wide variety of program semantics; assertion generation approaches must look beyond simple equality to capture these behaviours.

2.2.4 Assertion Equivalence

Most assertions can be constructed in multiple semantically equivalent forms. Semantically equivalent assertions generally fall into two categories: In the first, the assertion library provides multiple equivalent assertion methods (e.g., `equals` and `eq`) as a form of syntactic sugar for performing the same underlying check. In the second, developers make different stylistic choices that are semantically equivalent (e.g., `expect(arr.length).to.equal(1)` vs. `expect(arr).to.have.length(1)`). This kind of equivalence, with one choice being a specific assertion operator and the other being framed as an equality check, was the most commonly observed form of semantic equivalence.

To identify the prevalence of semantically equivalent assertions, we randomly selected 5% of each category's assertions from Table 1 and checked whether each assertion was categorized correctly or was actually an equivalent semantic form from another category. In this analysis we did not note any persistent misuse of 11 of the 12 assertion categories.

In contrast to other categories, equality-based assertions were often used in place of other more semantically-specific assertion operators. In total, 50.7% of assertions were framed as equality checks. However, our manual examination of the random sample of equality assertions revealed that 22.5% of these checks could be written with a more specific assertion operator. This reduced the prevalence of equality assertions from 50.7% of all assertions to 39.3%. Table 2 shows the distribution of equality checks that are actually semantic members of other assertion categories. We refined our static-analysis tool to reclassify instances of the seven types of specialized equality checks (listed in Table 2) as belonging to their respective semantic category and we re-ran analysis. The numbers and percentages reported in Table 1 reflect the results of this refined analysis, reporting the semantic intent of the analyzed assertions.

TABLE 2
Analysis of the uses of the equality assertion methods to determine what properties developers check with this operator. This shows that 22.5% of equality assertions in our study assert more semantically-specific behaviour.

Category	%	Example
Length	9.0%	`expect(val.length).to.equal(0);`
Boolean	8.1%	`expect(val).to.equal(true);`
Calls	2.4%	`expect(val.callCount).to.equal(1);`
Existence	2.2%	`expect(val).to.not.equal(null);`
Inclusion	0.5%	`expect(val.includes('foo')).to.equal(true);`
Type	0.2%	`expect(typeof val).to.equal('string');`
Numeric	0.0%	`expect(val < 0).to.equal(true);`

There are two main advantages to using a specific assertion operator rather than an equality-based operator (i.e., a call to `equals`). The first advantage is that the meaning of the assertion is more directly evident from the assertion statement itself; for example, the following two assertions are semantically equivalent, but the first more clearly encodes its intent:

```
// Equivalent assertions
expect(arr).to.contain('c');
expect(arr.indexOf('c') >= 0).to.equal(true);
```

Listing 1. Two equivalent assertions; the first uses a specific assertion operator while the second uses a more general operator.

The second advantage is that, upon assertion failure, the more-specific assertion can report a more-specific error message for the developer to act on. Consider the error messages given when the above two assertions fail; the more specific assertion is able to raise a more meaningful error message:

```
// Corresponding failures
AssertionError: expected [ 'a', 'b' ] to include 'c'
AssertionError: expected false to equal true
```

Listing 2. Error messages emitted when the two assertions in Listing 1 fail. The more specific assertion provides a more descriptive and actionable error message.

> □ Developers may write assertions for the same behaviour in a variety of ways; assertion generation approaches should be aware of these differences, while promoting consistency and specificity.

RQ1 Summary

Most test cases contain relatively few assertions (median=1) and these assertions are not complex. Assertions can be broadly classified into 12 high-level categories: 39.3% of assertions validate equality and the rest check specific behaviours, although semantically-equivalent forms are common.

3 Automatic Assertion Generation

Based on the findings about developer-written assertions from Section 2, we created a prototype tool called AutoAssert that automatically generates many common kinds of assertions. Assertion generation proceeds in four main phases, as depicted in Figure 3. First, the developer identifies a variable in a test case for which they want assertions; the test case is executed (twice), and the values assigned to the variable are recorded for each execution. Second, AutoAssert identifies the kinds of assertions that should be generated based on the values assigned to the variable

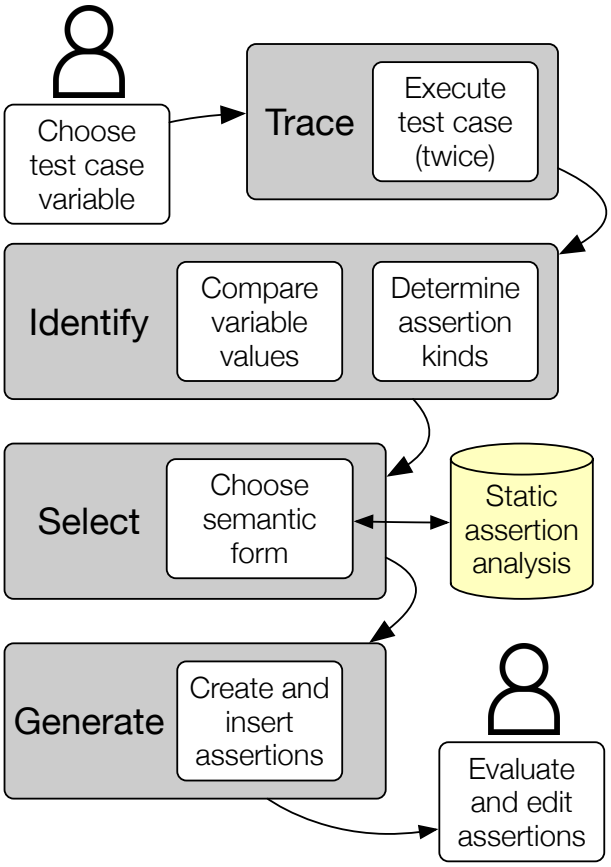

Fig. 3. Automatic assertion generation process. The developer initiates the assertion generation process by selecting a specific variable in a test case. AutoAssert traces the test case execution, identifies which assertions are appropriate, selects the best semantic form for the developer's project, and generates the proper assertion text and inserts it into the test case. The developer then reviews the assertions to ensure both that the assertion operators are appropriate and that the observed values match their expectations; they edit or remove any assertion that does not meet their expectations.

at runtime. Third, the most appropriate semantic forms are selected for the generated assertions, based on the norms of the project. Finally, the assertion text is generated and inserted into the test case for the developer to review. This review step is crucial, because AutoAssert presumes that the code under test is correct. Each of these phases is described below.

3.1 Design methodology

For our prototype implementation of AutoAssert, we chose to generate assertions for the EQUALITY, BOOLEAN, LENGTH, EXISTENCE, and TYPE and THROW categories from Table 1 as our dynamic approach seemed particularly amenable to these. Additionally, these categories have clearly observable properties that imply the exact expected values they should be validated against. One slight difference is the THROW category where JavaScript allows variables to be assigned as function pointers. If a variable is a function pointer that does not take arguments, AutoAssert can create appropriate assertions.

Some assertion categories are more challenging to support including INCLUSION, PROPERTIES, CALLS, NUMERIC, PATTERNS, and TRUTHINESS. INCLUSION and PROPERTIES both rely on knowledge of *which specific* property, element, or substring of the variable under test is the important one to validate and cannot be determined by the resulting value alone. Similarly NUMERIC and PATTERNS assertions both involve comparisons against a broader set of values where many possible ranges and patterns could be correct that AutoAssert does not have the ability to choose between (for example `expect(id).to.be.greaterThan(7.4)`). CALLS assertions check to make sure that a function has been called, rather than inspecting a value for correctness; it is not clear how one could predict when this kind of check is appropriate. Finally, although easy to implement, TRUTHINESS was the least seen category and we considered it to be largely redundant given that AutoAssert supports both the BOOLEAN and EXISTENCE categories.

3.2 Tracing Program Values

Although a variety of approaches could be used to determine the values of variables under test, AutoAssert uses a dynamic approach by executing an individual test case and recording the runtime values assigned to the variable selected by the developer.

```
it('Should generate v1 with options', function () {
  const options = {
    node: [0x01, 0x23, 0x45, 0x67, 0x89, 0xab]
  };
  const uuid = UUIDUtil.generate('v1', options);
});
```

Listing 3. An example test case. A developer would generate assertions by selecting `uuid` and invoking AutoAssert.

For example, for the test case in Listing 3, the developer selects a variable of interest (e.g., `uuid`) by right clicking on the variable and invokes AutoAssert. The tool invisibly injects observation code into the individual test case immediately below the statement containing the selected variable instance, executes the test case twice, and records the values assigned to the variable (`uuid`). After the test case is executed, the observation code is removed.[2] JavaScript and TypeScript provide native support for traversing the properties declared on a variable, enabling AutoAssert to easily examine the dynamic state of the variable after executing the code under test.

Listing 4 shows the recorded values of variable `uuid` for the two executions. Running each test case twice provides an opportunity to detect when the two recorded values differ (e.g., when assigned a timestamp or a randomly generated value like a UUID)[3]. Identifying the properties of a variable that change between executions decreases the probability that AutoAssert will generate assertions that are too strong. In this case, we can see that the runtime values of `uuid` in the two test-case executions (1) are both strings, (2) have the same length, but (3) contain different values. In the case that two recorded values are identical (which is the norm, rather than the exception), the value is recorded only once. For more complex variables (e.g., for objects),

2. For TypeScript projects, the project is incrementally compiled after injecting the tracing code, but before the test is run.

3. We hypothesize that non-deterministic tests will either be fully random, or have potential differences that are determined by external factors such as time, platform, or environment variables. Under this assumption, running the test case twice should be sufficient to detect differences.

the `value` property will contain the serialized object. To increase the options for generated assertions on instances of classes, AutoAssert includes with the serialized value of an object all method names as well as all property names and values[4].

```
[{
  type: 'string',
  value: '8b839680-e0e9-11ea-b5a6-0123456789ab',
  length: 36
},
{
  type: 'string',
  value: '232ab3a0-e0cd-11ea-b840-0123456789ab',
  length: 36
}]
```

Listing 4. An example of a final value recording, showing the identified type and value. In this case, the variable's values differ in the two executions.

When a variable changes between executions, AutoAssert only generates assertions on aspects of the variable that are common between executions. For example, in Listing 4, an existence assertion will be generated because the variable contained a value in both executions. Since the variable is a string in both executions, a type checking assertion will also be generated. As the variable value differs between executions, an equality assertion will not be generated.

3.3 Identifying Assertion Categories

Both test case executions usually return the same value for the variable under test. AutoAssert next identifies the most appropriate assertions for that value. While it would be straightforward to simply use the equality operator to create an assertion to compare the variable to its traced value AutoAssert tries instead to find a set of more-specific assertions in order to maximize the utility of the error message raised for any failing assertion, as detailed in Section 2.2.4.

In general, AutoAssert tries to generate three assertions, each of increasing strength. The first, weakest, kind of assertion checks for the existence of a value (e.g., `undefined` or `null`). This assertion is applicable to all variables in JavaScript and TypeScript test cases. The middle-strength assertion checks an attribute of the variable, depending on the runtime value of the variable. For an array, this would usually be its length. For all other variables, its type will be checked (e.g., `string`, `number`, `object`). For traces that involve a thrown exception, the `throws` operator is used to ensure an exception is thrown. Finally, the strongest kind of assertion usually performs an appropriate equality check (e.g., equals for primitive types, deep equals for complex objects or arrays) to ensure the actual value matches the expected value.

With respect to the last assertion, AutoAssert prioritizes specific assertions over generic equality-based assertions. To do this, the tool always selects non-equality assertion categories (from Table 1) if they are applicable to the given variable and only falls back on equality when more specific assertions are not appropriate (for example `expect(val).to.be.true` would be used in place of `expect(val).to.equal(true)` for boolean values). If the developer selects a variable of interest that does not have a clear value, such as a function or a promise, then only existence and type checks (and no equality check) are generated.

While generating assertions from several categories often results in 'extra' assertions, the additional assertions increase the specificity of the resulting error messages. For example Listing 5 shows three assertions for a simple object assigned to `val`. The first checks that the object `val` exists (e.g., is not `undefined` or `null`). The second ensures that `val` has the right type. The final check validates that `val` has the expected value. While only the last check is required (as a non-existent or wrongly-typed value would fail the equality check), the initial checks would produce more specific and easier to understand error messages should an assertion fail in a future test run, making it easier for the developer to diagnose the fault. These assertions are not meant to be absolute: it is expected that the developer will delete any assertion they deem redundant or too strict to suit their personal preference, or edit them to meet their needs.

```
const val = db.getPaper('2021-10-0442');
// Exist means neither null nor undefined
expect(val).to.exist;
// Typecheck similar to typeof operator
expect(val).to.be.an('object');
// Value equality
expect(val).to.equal({venue: 'tse', year: '2022'});
```

Listing 5. An example set of Chai assertions generated after a line of test code that returns a simple object.

3.4 Selecting and generating assertions

Given that developers can write assertions in many different forms, AutoAssert tries to increase the probability that the generated assertions will match developer expectations by choosing the assertion form that best matches those used by a given project. For example, Listing 6 shows how even a simple null-checking assertion can be written three different ways. While the first of these is the most specific and would be preferred in the absence of any other input, AutoAssert can optionally statically analyze the developer's project to determine if the project usually uses specific versions of equivalent forms for any assertion category.

```
expect(val).to.be.null;
expect(val).to.be.a('null');
expect(val).to.equal(null);
```

Listing 6. Semantically equivalent assertions for checking whether `value` is `null`.

To do this, AutoAssert scans all test cases in the project, and counts the usage of each assertion form for each assertion category. For example, for equality assertions, the analysis counts the usages of different forms of equality (e.g., `eq`, `eql`, `equal`, `equals`, `toEqual`, `strictEqual`, `deepEqual`, `deepStrictEqual`, `toStrictEqual`, `equalIgnoreSpaces`) to determine which is most used for checking equality in the existing test cases so that generated assertions can be as consistent as possible with prior developer-written assertions. This analysis only needs to be done once, but can be re-run at any time.

Selecting project-specific forms is optional; if the static analysis has not been run on the project (or there are not yet any test cases in the project), AutoAssert will select the most

4. Note that JSON's standard serialization of objects (`stringify()`) does not serialize method names.

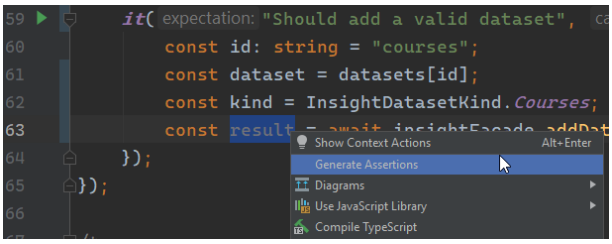

Fig. 4. Usage of AutoAssert through IntelliJ context menu, adding an option to the context menu when a selection is made.

common forms that were identified in the initial assertion equivalence analysis (Section 2.2.4).

Generating assertion text is straightforward once the assertion category, equivalent form, and order of an assertion is determined. The generated text is a multi-line string that is injected into the developer's test code immediately following the program statement that contains the variable instance on which they invoked AutoAssert.

3.5 Evaluating assertions

Since AutoAssert is designed as an interactive human-in-the-loop system, the developer finalizes their test case by evaluating whether each generated assertion matches their expectations. This step is crucial because assertions that do not match their expectations could uncover unexpected behaviour in the code under test, or could represent assertions that differ from how the developer would like to evaluate the current behaviour. The developers have three broad ways to audit each assertion: (1) For any assertion that matches their expectations, they can *leave* the assertion text in the test case. (2) Developers can *edit* the text (e.g., to make an assertion more specific or more broad) by modifying the assertion text directly. (3) The developer can *delete* any assertion that does not evaluate behaviour a developer believes is appropriate for their test case.

AutoAssert includes a rationale statement, in the form of a comment, inserted into the code along with each assertion to explain what behaviour the assertion is validating. These comments can be seen in Listing 5. These comments can be disabled with a tool preference if developers find them to be overly verbose.

3.6 Implementation

Our approach is amenable to any language that supports dynamic variable introspection. The AutoAssert prototype supports both JavaScript and TypeScript. The JavaScript ecosystem supports a large number of assertion libraries; we selected the Chai assertion library, although other assertion libraries could be supported.

To embed AutoAssert into a developer's normal workflow, the prototype was implemented as an extension to the IntelliJ IDE[5] because of its strong support for extensions, wide user-base, and support for both JavaScript and TypeScript. Figure 4 shows AutoAssert being invoked from within IntelliJ. An online demonstration of the AutoAssert plugin is online[6] and the source code the entire AutoAssert IntelliJ implementation is available online[7].

AutoAssert is invoked with a context menu option "Generate Assertions" that appears whenever a developer right-clicks on a variable in a test case in their IDE. AutoAssert invisibly injects the observation code, runs the test case (including the injected code), generates and inserts the assertions into the test case for the developer to inspect, and removes the injected observation code. The time required for this process depends on the time it takes to run the test case itself through the IDE; AutoAssert itself has no meaningful overhead.

4 Evaluating Assertion Correctness

To evaluate the quality of the assertions generated by our prototypical AutoAssert tool, we performed an empirical simulation comparing our generated assertions against those written by developers in real software projects. To do this, we investigated:

RQ2: Can AutoAssert generate assertions similar to those written by developers?

The goal of this simulation is to see how consistently AutoAssert can generate an assertion in the same category as a developer-written assertion for the same variable in the same program statement. Being able to consistently produce assertions similar to those written by developers is crucial for making an assertion generation tool that developers could use and trust.

4.1 Methodology

To perform our empirical simulation, we selected 10 open source projects with developer-written test suites. Projects were selected from our final list of candidate projects as discussed in Section 2.1, sorted by stars. We then chose the first 10 projects whose test suites we were able to execute. We identified all developer-written assertions in the passing test cases from the 10 selected projects for a final list of 1,335 assertions. For each of these assertions we extracted the element under test (assumed to be the left-hand side of the assertion) and used this as input to trigger assertion generation with AutoAssert. For each of these, AutoAssert generated one to four assertions.

We then evaluated whether the assertions AutoAssert generated for a variable matched those written by the developer. One author manually compared the assertions to verify whether the semantics of the developer-written assertion was replicated by the generate assertions (accounting for equivalent forms as described in Section 2.2.4). A second author independently examined a random sample of 20% of the inspected assertions to verify the consistency of the categorization process. The independent rater agreed on the categorization for all 267 of the 267 randomly inspected assertions, suggesting that assigning an assertion to its category can be reliably performed.

Listing 7 shows an instance of a category match. In this case one of the generated assertions happens to be identical,

5. https://www.jetbrains.com/idea/

6. Online demonstration: https://youtu.be/w1MoeZxfJko
7. Source: https://github.com/LucasZamprogno/AutoAssert

TABLE 3
Breakdown of the dynamic generation results. AutoAssert almost always generates an assertion of the same category (for the supported assertion categories).

Category	% of Sample	Total	# Correct	% Correct
Equality	60.22%	804	796	99.00%
Boolean	7.57%	101	101	100.00%
Existence	7.57%	101	101	100.00%
Type	4.87%	65	64	98.46%
Length	4.42%	59	57	96.61%
Throw	0.67%	9	9	100.00%
Numeric	8.84%	118	0	0.00%
Truthiness	4.34%	58	0	0.00%
Properties	0.75%	10	0	0.00%
Inclusion	0.60%	8	0	0.00%
Calls	0.07%	1	0	0.00%
Patterns	0.07%	1	0	0.00%
Total	**100.00%**	**1,335**	**1,128**	**84.49%**

but this would also be considered a match if any other equality-based assertion method were used. Listing 8 shows a category miss, where the original assertion performs a type-based check, whereas our assertion checks that the value is exactly `null` as this was the value `error` actually had assigned to it when the test executed.

Essentially, this evaluation examines whether *any* of the (at most four) assertions match the assertion written by the developer. This evaluation was not intended as a *top-k* comparison as AutoAssert intentionally orders generated assertions from weakest to strongest, and intentionally generates additional assertions that we believe improve the quality of the assertion error messages, even if these extra assertions are not exactly what the developer would write themselves. For the generated assertions in Listing 7 we could have configured AutoAssert to only generate the final assertion, which matches the developer-written assertion, but (as described in Section 3.3) we generated additional assertions improve the understandably of a future assertion failure. In practice this means that the last generated assertion is the most likely to match the form written by the developer as it is the strongest.

```
// Original assertion:
expect(data).to.deep.equal(['test1', 'test2']);
// Generated assertions:
expect(data).to.exist;
expect(data).to.be.a('array');
expect(data).to.have.length(2);
expect(data).to.deep.equal(['test1','test2']);
```

Listing 7. Example of generated assertions containing the same category (equality) as the original. This is considered a hit.

```
// Original assertion:
expect(error).to.not.be.instanceof(Error);
// Generated assertion:
expect(error).to.be.null;
```

Listing 8. Example of generated assertions that did not contain the same category (`type`) as the original assertion. This is considered a miss.

4.2 Results

Table 3 shows the result of this evaluation. The AutoAssert prototype generates assertions only for the categories in the top half of the table. 85% (1,139 of 1,335) assertions in this study fall into the six supported assertion categories. Of these, AutoAssert generated corresponding assertions in the same category 99% of the time (1,128 of 1,139 assertions). This resulted in an overall accuracy of 84.5% (1,128 of 1,355 assertions) for the current AutoAssert prototype.

One type of assertion that is particularly challenging to support is an assertion about what a value is *not*. Listing 8 shows an example of a developer-written assertion that a variable should not be instantiated with an `Error` object. Assertions such as these are challenging because the space of what a value is *not* is infinitely large. To understand the prevalence of negated assertions, we checked our results for the presence of the `not` modifier or methods such as `notEquals`: Of our successful assertion generations, 17 of the 804 equality assertions, one boolean assertion, and one length assertion all contain a negation. There are also 23 existence assertions with `not` modifiers, however AutoAssert is able to generate these correctly, so this type of negated assertion is not a concern.

Another failure mode is assertions where the element under test is equal to a certain named function, such as:

```
expect(global[methodName]).to
  .equal(ORIGINAL_DSL[methodName])
```

Generating such an assertion would require being able to identify the name of the function returned and the proper scoping in order to reference the function in an assertion.

Finally, as discussed in Section 3.2, assertion generation is more complicated when the variable under test has different values in different test runs. In our sample, 40 of the 1,335 (3%) assertions we generated had elements under test that were different when the test was run a second time. AutoAssert was able to detect these differences by executing each assertion twice and refrained from generating value-based assertions.

This evaluation shows that for most assertions of interest to developers, AutoAssert generates assertions of the appropriate category. Even if AutoAssert assertions do not exactly match developer-written assertions (especially with respect to variable values), the generated assertions are guaranteed to *pass* because assertion values are derived from variable values observed at runtime. This consistency is important because in the context of a developer-invoked tool, if a developer lacks confidence that the tool can produce the types of assertion they want with valid values, they may forego using the tool altogether.

RQ2 Summary

Using a heuristic-based dynamic approach to assertion generation, AutoAssert can consistently produce assertions in the same category as those originally written by developers. Across over 1,000 developer-written assertions, AutoAssert reproduced the original assertion category 99% of supported assertions, with an overall accuracy of 84.5%.

5 User Study

Since AutoAssert was able to generate appropriate assertions for 84% of cases, we next performed a user study with developers to learn how they perceive the generated

assertions. This evaluation was important because while assertion generation tools have been evaluated empirically, they have rarely been evaluated interactively with real developers. Specifically, we sought to answer the following two research questions:

RQ3: How do developers write assertions in practice?

RQ4: Do developers find the assertions generated by AutoAssert valuable?

These questions investigate whether the assertions generated by AutoAssert are valuable to real developers.

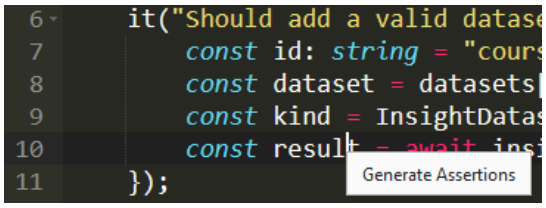

Fig. 5. Usage of AutoAssert through the web interface created for the user study, replicating the context menu behaviour from the initial plugin version.

5.1 Methodology

Participants for our user study were recruited primarily through various software-development related forums on Reddit.[8] 23 individuals, 21 men and 2 women, participated in the study in some capacity, with 17 completing all of the study's tasks and both surveys. The study took 20–30 minutes to complete and 4 of the developers won $50 gift cards through a raffle at the study conclusion.

The study was completely browser-based and began with a pre-task survey; the participants next used a web-based version of AutoAssert to generate assertions for eight test cases; and the study concluded with a post-task survey. This online format was not our preferred modality, but in response to COVID-19 we decided this was the prudent way to move forward and engage engineers.

5.1.1 Pre-task survey

The pre-task survey collected participant consent and gathered relevant demographic information including years of professional programming experience, years of experience writing tests, and familiarity with JavaScript. The goal of the pre-task survey was to answer **RQ3** and learn about the processes participants follow as they write their tests and assertions; participants were asked the following questions:

- PRE-1: Do your tests include (a) no assertions, (b) assertions for pre-conditions, (c) assertions for post-conditions, (d) both pre- and post-conditions.
- PRE-2: What process do you follow when writing assertions?
- PRE-3: Do you use any external tools for test or assertion generation?

At the conclusion of the pre-survey, participants were automatically forwarded to our browser-based version of AutoAssert.

5.1.2 AutoAssert tasks

To answer **RQ4** we sought to have participants use AutoAssert and evaluate the generated assertions for real test cases taken from real systems. The browser-based AutoAssert is based on the same code backend as the IDE version of the tool, but with a browser-based code editor[9]. The UI for the tool was otherwise the same: participants could view the test and associated source file, they could invoke AutoAssert with a context menu by clicking on the variables they wanted assertions for, and AutoAssert would inject its generated assertions into the browser-based editor for the participants to manipulate as they thought appropriate. The UI for the browser-based AutoAssert is shown in Figure 5.[10]

For this portion of the study, participants used AutoAssert to generate assertions for a series of test cases. When presented with generated assertions, the participants were free to delete, modify, or add any assertions they wanted until they felt the assertions were appropriate for the test case. Participants were also able to use code comments if they wanted to provide specific feedback about the assertions. Participants performed one training task and seven experimental tasks. The training task was a simple test case along with comments that showed the participant how to invoke AutoAssert and guided them through a sample task.

Five of the seven non-training tasks were taken from two open source projects: Typeset,[11] which is a string manipulation library for replacing ASCII characters with more visually appealing Unicode characters, and Nock,[12] which is a server test mocking framework. Each task consisted of a real test case selected from one of these projects but with the assertions removed. To allow participants to concentrate on understanding the test code instead of the product code, we added a short comment to each test case summarizing the intended functionality being tested. Project order was randomized, as was test order within a project; however, test cases from the same project were always presented together so that participants did not have to context switch between projects.

We also included two 'poison pill' tasks that appeared between the two randomized blocks of tasks. These tasks generated assertions that were intentionally poor, to evaluate how developers reacted to poor automation. Specifically, for these tests, AutoAssert generated strict equality checks on code that produced unique results on every run (i.e., AutoAssert refrained from running the test cases twice to check for differences in elements under test). We devised these two tasks to ascertain whether participants would modify poorly generated assertions or comment on the challenges in the post-survey. These poison pill tasks were meant to check that participants were reading and thinking about the assertions as intended, and not immediately moving to the next task to finish the study faster. For all tasks, the browser-based tool recorded the state of the code editor when the

8. Specifically, `r/javascript`, `r/devops`, `r/softwaretesting`
9. Based on Ace https://ace.c9.io
10. An anonymized, log-free version of the complete browser-based AutoAssert implementation and all experimental tasks can be found at https://se.cs.ubc.ca/AutoAssert/ (VM may take a moment to start).
11. https://github.com/davidmerfield/Typeset
12. https://github.com/nock/nock

participant progressed to the next task (by tapping 'next' in a bar on the bottom of their screen). After completing all eight tasks participants were automatically forwarded to the post-task survey.

5.1.3 Post-task survey

The post-task survey acted as a debrief for the AutoAssert tasks. Participants were asked the following questions:

- POST-1: Would you use a tool like AutoAssert if it were integrated with your development environment?
- POST-2: How did you find AutoAssert's runtime performance?
- POST-3: In terms of assertion quality, how did the generated assertions compare with those you would write manually?
- POST-4: Were the automatically generated assertions better or worse than what you would normally write?
- POST-5: Can you foresee scenarios where assertion generation may fail?
- POST-6: Do you have any suggestions to improve the automatically-generated assertions?

The first three questions were based on a Likert scale while the rest were open ended.

These questions sought to gain further insight into **RQ4** as well as record how well developers believed they had performed on the study's AutoAssert tasks.

5.2 Results

In this section we discuss the results from our user study, including the pre-survey about real-world, test-development behaviour, the study's assertion-generation tasks, and post-survey feedback about AutoAssert.

5.2.1 Pre-task survey.

Twenty-three participants completed our pre-survey (91% male). Of these, 74% identified themselves as professional engineers and had an average of 11.7 years of development experience. 74% of participants said they were at least moderately familiar with JavaScript. Seventeen participants completed the AutoAssert tasks and post-survey in full.

Kinds of assertions (PRE-1). Two participants (9%) stated they did not typically include assertions in their tests, beyond the tests' built-in ability to detect thrown errors. Eleven participants (48%) claimed to include assertions as post-conditions and ten participants (43%) claimed to use assertions as both pre- and post-conditions in their test cases.

Assertion writing process (PRE-2). Participants reported to employ a wide variety of processes when creating their test cases and assertions. Although for simple unit tests developers may have some assertions in mind when starting their test case, for more complex test cases one participant said, *"If [the code under test] is something more complex, I run the code and then verify that the result makes sense before writing assertions."* This notion of checking the output before writing the assertions arose frequently, *"I would run the code inspect the output and then write assertion against it"*, *"I inspect the output before writing my assertions"*, and *"inspect the actual output after running the test"*.

Ultimately, although most participants (91%) reported that they often knew in advance which assertions they would write, a surprising number (65%) mentioned running the code under test and inspecting its output in one form or other. Taken together, these comments suggest that the participants' approaches to writing assertions fit well with the process AutoAssert embodies, as these developers often have (at least initial) implementations before writing their test cases and it is not uncommon that they inspect program output before writing assertions.

Existing tools (PRE-3). No participants reported using any existing test generation or assertion generation tools. One concern with such a tool was the overhead associated with learning how to use such a tool and whether it would fit with their typical test-writing processes. Another concern was around whether the assertions would actually be accurate for the code under test.

RQ3 Summary

Developers often write their test cases and assertions after developing the code under test. They frequently run their new test cases and inspect the values returned by the code under test while creating and debugging their assertions.

5.2.2 AutoAssert tasks

Seventeen participants completed all of the study's AutoAssert tasks, of which fourteen of the seventeen participants indicated that the 'poison pill' tasks were problematic (they either changed the assertions or made inline comments), suggesting a high level of engagement with the AutoAssert output. In our analyses below, we exclude the three developers who did not notice these problems (for a remainder of fourteen participants).

Five participants left all of the assertions generated by AutoAssert in place for the five good tasks, suggesting that they agreed that the assertions generated by AutoAssert were appropriate. For the nine participants who changed the assertions, the most common behaviour (performed by six participants) was to remove the less strict existence and type-checking assertions in places where they believed (correctly) that an equality check would also catch these faults.

Two participants left comments on Nock tests involving server responses, noting that they would create assertions for both the response code and response body, rather than create assertions for only one or the other. Such comments reflect on the structure of the task (as it was setup to evaluate the response code, not the response body) rather than on the tool itself. One participant deleted the strict equality check from the string-manipulation tasks, leaving the existence and type checks. One participant extracted a literal value, which appeared in both the test setup and the generated assertion, into a variable for cleaner code. Three participants made changes to the expected values that did not match the program behaviour in at least one test, presumably misunderstanding the intended behaviour of the code under test or believing the code to be incorrect.

AutoAssert task summary. Ultimately, the fourteen engaged participants generated a combined total of 210

assertions using AutoAssert for the non-poison-pill tasks they performed. Of these assertions, they "approved" 174 (83%) as being correct by leaving them in the code and "rejected" 36 (17%) by removing or modifying them. Figure 6 shows an instance of a participant removing two of the three assertions generated during a task.

5.2.3 Post-task survey

After using AutoAssert for eight tasks, participants provided a wealth of feedback. Quantitatively, 15/17 (88%) of participants expressed an interest in using a tool like AutoAssert if it were available to them (POST-1). The runtime performance of a tool like AutoAssert can be a concern given the tight feedback loop developers perform while iterating on their tests. 16/17 (94%) of respondents found the tool performance to be acceptable or better (POST-2). Naturally, we were most interested in how developers perceived the quality of the assertions. Only 4/17 (24%) of participants found AutoAssert's automatically-generated assertions to be *somewhat worse* than those they would write themselves (POST-3). More details about these responses can be seen in Figure 7.

AutoAssert assertion quality (POST-4). The participants who felt that the assertions could be improved provided opposing views as to what 'better' means for assertion quality. Four of the participants suggested that early, less-strict assertions were functionally captured by the later more-strict assertions, were unnecessary, and should be removed. Other participants appreciated the increasing specificity of the generated assertions and worried that the most-strict assertions might be overly specific, preferring to remove the latter. For example, one participant noted that *"They do cover cases that I might miss (e.g., the exists checks)"* where another stated that *"I found the exists and type assertions redundant. I like that they might help debug the cause of a bug but the diff given when the test fails should make those cases obvious to spot with the value assertion alone."* This suggests that a one-size-fits-all Assertion Generation approach might not be feasible and that some kind of personalization might be warranted to accommodate divergent assertion-writing strategies.

Participants did note that AutoAssert produces assertions that they might not have created themselves but that they thought were valuable, *"It would generate assertions that I might be too lazy to write myself"* and *"They were very thorough, which encouraged me to include more of them."*

```
it('matches a query string', async () => {
      const scope = nock('http://example.test')
        .get('/')
        .query({ foo: 'bar' })
        .reply();

      // Target variable is statusCode
      const url = 'http://example.test/?foo=bar';
      const { statusCode } = await got(url);
      -expect(statusCode).to.exist;
      -expect(statusCode).to.be.a("number");
      +expect(statusCode).to.equal(200);
});
```

Fig. 6. Three assertions generated by AutoAssert during a task. When evaluating the generated assertions, the participant removed the first two assertions (as shown in red text) but kept the final, most specific, assertion (as shown in green text).

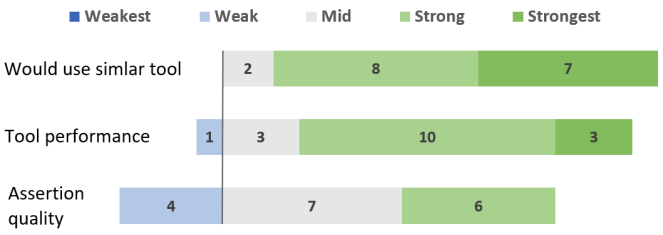

Fig. 7. Breakdown of participant responses to the three Likert-scale questions from the post-task survey (POST-1, POST-2, and POST-3).

Encouragingly, many participants commented on the poison-pill tasks.[13] These participants noted that assertion generation would not be appropriate for test cases that have random or changing values (POST-5). Fortunately, the real AutoAssert tool does handle these situations, and it would not have produced failing assertions for these two tasks if all its features had been enabled. Participants also expressed concern about other scenarios not included in the tasks (POST-5): One participant noted that it would be hard to handle cases where only a substring of a result (part of our Inclusion category) was the only important part of the return value. Multiple participants noted that deep equality checks would be decreasingly appropriate as objects or arrays under test became larger.

Improving assertion generation (POST-6). Many participants had general advice for improving assertion-generation tools. One participant suggested having an explicit configuration option to adjust the strictness of the generated assertions, such as a choice between *"deep equals vs. some elements are equal vs. lengths are equal vs. length is at least vs. length is non-zero"*. Multiple participants felt that the generated assertions should try to adhere to the coding style and conventions of existing test cases (e.g., for quotes, line lengths, and indentation). One participant, rightly, was worried that automated assertion generation *"may offer the developer a false sense of security"*. We fully agree: we believe that it is important that assertion-generation systems be human-in-the-loop, so that the developer can carefully examine the automatically generated assertions and ensure the values they check are actually correct (i.e. that the assertions test the right behaviour, and do not create passing assertions for incorrect behaviour).

RQ4 Summary

Developers found the assertions created by AutoAssert to be broadly useful at validating program behaviour for the test cases they evaluated. We hypothesize that additional personalization (e.g., for strictness and style) could address the majority of concerns developers identified with the generated assertions.

13. Participants did not know that these tasks generated intentionally 'poor' assertions.

6 Discussion

In this paper we have confirmed that most test cases have few assertions and those assertions tend to be uncomplicated (Section 2), and that AutoAssert can generate the majority of these automatically (Section 4). We have also found that assertion generation can fit within a developer's testing workflow and that developers respond positively to generated assertions (Section 5). That said, AutoAssert has several limitations, and developers have made concrete suggestions for future improvements.

6.1 Adaptations to Developer Feedback

Previously, Section 3.4 described how AutoAssert detects which assertion forms to use in a test suite. However, developers expressed a desire to be able to explicitly overrule the detected forms, and we have added this capability to the AutoAssert customization page. For each assertion category, developers can optionally choose which specific form of assertion they prefer, overruling the form chosen by AutoAssert's static analysis. To further address concerns around superfluous assertions, AutoAssert now includes a verbosity option enabling developers tho choose whether minimal or more verbose assertions should be generated.

6.2 Improving Readability with Generated Assertions

While not the focus of our study, we formed several observations while reading through large numbers of sampled and generated assertions about how assertion generation tools could improve test case readability. One problem observed in our sampled assertions was a surprising number of instances where developers had written assertions "backwards". That is, the value under test is placed as the expected value, and vice versa. For example assertions such as `expect(true).to.equal(result)`. While the functional correctness is often maintained in these cases, assertion readability suffers. Generated assertions would not make this mistake, and would improve readability through consistency. Although there are many different assertions methods and semantically equivalent forms for the same behavioural check, generated assertions would be consistent in their selection of assertion forms for a given behaviour. Additionally, when multiple assertions are produced they will always be produced in the same order in terms of strictness without developer intervention. Much like clean code is desirable for easier understanding and maintenance, clean assertions similarly benefit test code.

6.3 Limitations and Future Work

There are a few areas where our heuristic approach for generating assertions falls short.

Properties within complex objects. When asserting behaviour on complex objects, developers often assert on a specific property (or subset of properties) about the object. Although a developer could choose instead to save that specific property in a variable and then assert on that variable, it would be more natural for their workflow if a tool could suggest or determine the correct property. One extension that we aim to add is interactive support that would allow users to explore and choose which properties to assert on, for languages with static class definitions. This would improve the selection of assertion targets and the specificity of the assertion methods.

Richer assertion categories. Currently AutoAssert supports assertion categories like Existence, Type, and Equality well. However categories such as Ranges and Inclusion are currently not supported. Expected values for these categories are not directly inferrable from observed values as they could require isolating a particular subelement of interest, or extrapolating a range of permissible values. Improvements in this area would likely require stepping out of the bounds of simple heuristics. One direction could be to combine the observed runtime values with a learning approach like used by Watson et al. [10].

Assertion evolution when tests fail. Maintaining test suites requires considerable developer effort [14]. AutoAssert could be extended to help developers update their tests when assertions fail to account for new dynamic behaviour by presenting them with new assertions for their consideration, thereby easing the process of evolving their test cases.

6.4 Threats to validity

Several threats to validity should be considered given our experimental design.

External validity. The kinds of assertions developers write may be influenced by their choice of language and framework. In this paper we have focused on JavaScript and TypeScript and our findings may not hold for other languages. Similarly, the projects we selected for our AutoAssert user study and quantitative analysis may not be representative of all test suites and test cases, although we tried to reduce this threat by examining a broad selection of 105 projects.

Specifically, the choice of language and framework may impact the relative weights and categories given in Table 1. For example we would expect the percentage of type checks to be substantially lower in statically typed languages such as Java, compared to the dynamically typed JavaScript where developers do not have built-in type checking at compile time. However we do not believe language or library selection weakens other aspects of AutoAssert: the motivations behind developer-in-the-loop dynamic assertion generation should still be broadly applicable.

Our generated assertions are also molded by the language we chose to support. JavaScript is well suited to creating and comparing literal values. Prior work has shown how these limitations can be overcome: The Orstra system demonstrated that static analysis can find appropriate observer methods to view private class fields, and suggest sequences of method calls to create class objects for use as expected values in Java [15].

Internal validity. Our empirical study of developer-written assertions in Section 2 did not consider assertions within helper functions as they could not be attributed to individual test cases. Additionally, the method in Section 4 considered AutoAssert generated assertions to be 'correct' if they matched the same category as assertions written by developers. It is possible that our evaluation results would have reported fewer matches if we instead had directly

compared the semantics of the generated and developer-written assertions. We tried to reduce this threat by engaging industrial developers to use the tool and evaluate generated assertions for real test cases. The tasks our participants completed with AutoAssert were based on projects and tests with which they were not familiar; this is unusual as test-writers are typically familiar with the code they are validating. As such, some study participants may have been more deferential to automated tools than they otherwise might be. Since the focus of the user study was on the assertions themselves rather than the test outcome, participants might have behaved differently if they had run the test cases in their own familiar environments. Unfortunately, because of the COVID-19 pandemic, we felt obligated to pivot to an online user study.

The quantitative study in Section 2 and simulation study in Section 4 both examine the assertions written by developers in their own test cases. Our analysis did not seek to determine whether these developer-written tests were unit, integration, or system tests. While our primary focus was to support unit testing, we do not see specific barriers to AutoAssert helping developers generate assertions for other categories of tests, as long as the code under test can be programmatically invoked and some output variable examined for correctness, although this bears future investigation.

7 Related Work

Several works have investigated generating assertions for a variety of tasks. Support for generating assertions is motivated by the fact that developing test oracles is expensive and even human-facing approaches require innovation to reduce their costs [6]. Recent work has further confirmed that developer-written test-case assertions need to be strengthened to better detect faults [7]. Static approaches have the benefit of being relatively fast, at the risk of generating assertions that may not pass when executed. Dynamic approaches generate assertions that are more likely to pass (unless they explicitly seek out failing assertions as a part of fault identification). The accuracy of generated assertions matters, if the goal of the approach is to have developers examine failing assertions.

Most tools have been evaluated empirically, and thus far none have been evaluated with developers, despite being developer-invokable tools. A summary of the most closely-related prior work is provided in Table 4. Unfortunately, given the differences in approach, programming language, and tool intent, a direct comparison of the accuracy of these approaches is not possible. Compared to prior work, AutoAssert was uniquely designed to work interactively with the developer as they created their individual unit tests. In this way, AutoAssert seeks to reduce development friction associated with writing unit tests by enabling the developer to concentrate on invoking the code-under-test and letting the tool generate reasonable assertions for that code [16]. Our user-based validation is also unique as it provides the first evidence that developers can both use, and value, the assertions generated by our developer-in-the-loop approach.

TABLE 4
High-level comparison to prior approaches whose main focus is generating developer-facing test case assertions.

Tool	Method	Evaluation	
		Empirical	User
Atlas [10]	Static	Yes	No
UnitPlus [8]	Static	Yes	No
Obsidian [9]	Static	No	No
Eclat [11]	Dynamic	Yes	No
ZoomIn [12]	Dynamic	Yes	No
AutoAssert	Dynamic	Yes	Yes

The Atlas system by Watson et al. applies machine learning to assertion generation [10]. Atlas is trained on the JUnit tests of thousands of Java projects containing hundreds of thousands of assertions. These tests are also abstracted to improve pattern finding. The system was evaluated by removing assertions from a sample of tests excluded from the training corpus and seeing if Atlas could recreate them. For each removed assertion, Atlas achieved a 31% accuracy with its first generated assertion, and nearly 50% accuracy for the original assertion being replicated in the top five generated assertions. Atlas achieves reasonably high accuracy without the need for dynamic analysis. However without runtime information, the generated values cannot be guaranteed to match their actual values.

The UnitPlus tool recommends test and assertion code for developer-selected methods [8]. UnitPlus performs a static analysis of the class under test to determine which methods from the class are state-modifying and which are observers. UnitPlus then attempts to construct a series of state-modifying method calls to configure an object for use in an assertion. Similar to AutoAssert, UnitPlus is a developer-in-the-loop solution in which the developer selects an element under test to produce both test code and assertion code. Through an evaluation with four libraries, UnitPlus was shown to identify 187/314 (60%) state-modifying methods, which is how the tool generates assertions. However without dynamic information, any non-trivial expected value would be need to be provided by the developer.

Eclat leverages invariant detection to generate human-readable assertions for generated test cases [11]. Leveraging the Daikon invariant detection system [17], Eclat observes runtime pre- and post-conditions over a series of test executions. These invariants can then be used as pre-condition and post-condition assertions in future tests. Eclat's goal is to identify fault-revealing inputs and it meets this goal with an evaluated precision of 59% and a false-positive rate of 12%. The ZoomIn tool also leverages Daikon to identify assertions suited to locating faults and to note which assertions may be faulty. Additionally, invariants may inform future developer-written tests [12]. For example if an invariant appears to be unexpectedly restrictive, this may be indicative of an insufficient variety of test inputs. ZoomIn identifies 50% of fault-inducing inputs by examining only 1.5% of generated assertions. Neither Eclat or ZoomIn have been evaluated with users.

Obsidian is another instance of a developer-in-the-loop system to support developers in writing test cases [9]. Obsidian can setup test cases and structure the test cases with helper methods, leaving the developer to specify specific assertion values. Some values can be guessed using statically accessible defaults, such as specific constructor fields, to provide some tests with functional generated assertions. As such, Obsidian provides complementary functionality to AutoAssert by providing static test setup but requiring developers to manually determine correct assertion values. Unfortunately, these assertions have not been evaluated in terms of correctness or developer utility.

Test-suite generation approaches also generate assertions to act as oracles for the generated test suites. The Orstra test-generation tool combines state-modifying and observer-method analysis with dynamic runtime information [15]. With a focus on automatically generating test suites, Orstra executes tests and collects all object and observer-method states within the test as it runs. EvoSuite automates test-suite generation, whereby test-body code is created through a mix of methods including symbolic execution and an evolutionary approach [18]. EvoSuite uses mutation testing to select the assertions that are most likely to reveal faults, providing high-quality assertions at the cost of long runtimes associated with mutation testing. Similairly, Randoop also generates complete regression test suites [19] . In contrast, AutoAssert fills a different use case by assisting developers who seek to create or modify an individual test case rather than generate an entire suite. While test generation approaches tend to generate whole test suites, AutoAssert is designed to be used interactively to help developers build their test suite assertions within their normal testing process.

8 Conclusion

Assertions play a fundamental role in automated test cases as they check that the code under test behaves as expected. While assertions are numerous, they are also somewhat repetitive and have well-defined structure making them amenable to automatic generation. In this paper, we investigate how developers write assertions in practice through an empirical study and introduce AutoAssert, a tool to interactively generate assertions for developers as they write their test cases. AutoAssert is a human-in-the-loop system: a developer explicitly invokes the tool to generate assertions for a specific variable they want validated in a test case; they can then use their expertise to ensure both that the generated assertions match their expectations and that the asserted values are correct. In an empirical study of over 1,000 assertions, AutoAssert was able to generate assertions similar in purpose to those written by developers 84% of the time. Through a user study, developers reported that the generated assertions were broadly useful and many developers wished to use the tool for their own projects, showing developer interest in automated support for assertion generation.

References

[1] T. Xie, N. Tillmann, and P. Lakshman, "Advances in unit testing: Theory and practice," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 904–905.

[2] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 105–115.

[3] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien, "Rotten green tests," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 500–511.

[4] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM (CACM)*, vol. 41, no. 5, pp. 81–86, May 1998.

[5] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 214–224.

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *Transactions on Software Engineering (TSE)*, vol. 41, no. 5, pp. 507–525, May 2015.

[7] L. Jia, H. Zhong, and L. Huang, "The unit test quality of deep learning libraries: A mutation analysis," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 47–57.

[8] Y. Song, S. Thummalapenta, and T. Xie, "UnitPlus: Assisting developer testing in Eclipse," in *Proceedings of the Eclipse Technology Exchange (eTX)*, 2007, pp. 26–30.

[9] J. Bowring and H. Hegler, "Obsidian: Pattern-based unit test implementations," *Journal of Software Engineering and Applications (JSEA)*, vol. 7, no. 2, pp. 94–102, February 2014.

[10] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, pp. 1398–1409.

[11] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 504–527.

[12] F. Pastore and L. Mariani, "ZoomIn: Discovering failures by detecting wrong assertions," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 66–76.

[13] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, March 1990.

[14] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of Java projects using continuous integration," in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 821–830.

[15] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2006, pp. 380–403.

[16] N. Bradley, T. Fritz, and R. Holmes, "Sources of software development task friction," *Empirical Software Engineering Journal (EMSE)*, vol. 27, no. 7, p. 34 pages, December 2022.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming (SCP)*, vol. 69, no. 1-3, pp. 35–45, December 2007.

[18] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 416–419.

[19] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proceedings of the International Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, 2007, pp. 815–816.