

# **Automated Human-in-the-Loop Assertion Generation**

by

Lucas Alan Zamprogno

B.A., The University of British Columbia, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL  
STUDIES

(Computer Science)

The University of British Columbia  
(Vancouver)

December 2020

© Lucas Alan Zamprogno, 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Automated Human-in-the-Loop Assertion Generation**

submitted by **Lucas Alan Zamprogno** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Science**.

**Examining Committee:**

Reid Holmes, Associate Professor, Computer Science, UBC  
*Supervisor*

Ronald Garcia, Associate Professor, Computer Science, UBC  
*Supervisory Committee Member*

# Abstract

Test cases use assertions to check program behaviour. While these assertions may not be complex, they are themselves code and must be written correctly in order to distinguish between passing and failing test cases. I assert that test assertions are relatively repetitive and straight-forward, making their construction well suited to automation; and that tools can reduce developer effort (and simultaneously improve the quality of the assertions in their test suites) by automatically generating assertions that the tester can choose to accept, modify, delete, or augment.

Such a tool can fit into a developer workflow where tests are frequently written alongside runnable source code. I examined 33,873 assertions from 105 projects and identified twelve high-level categories that account for the vast majority of developer-written test assertions, confirming that test assertions are fairly simple in practice. To assess the utility of my human-in-the-loop assertion generation thesis, I developed the AutoAssert framework, which generates typical assertions for test cases written for JavaScript code. AutoAssert uses dynamic analysis to determine both which assertions to generate and what values they should verify. The developer can choose to accept, modify, delete, or add to the set of generated assertions. I compared assertions generated by AutoAssert to those written by developers and found that it generates the same kind of assertions as written by developers 84% of the time in a sample of over 1,000 assertions. Additionally I validated the utility of AutoAssert-generated assertions with 17 developers; these developers found that the majority of generated assertions were useful and expressed considerable interest in using such a tool/approach for their own projects.

# Lay Summary

When writing software systems, developers often rely on automated tests to ensure their software is functioning correctly. One core component of a test is the assertion, which checks a programs actual behaviour against its expected behaviour. There have been numerous approaches developed to help automate the process of creating tests, and their assertions. This thesis presents a new prototype tool called AutoAssert designed to fit into a developer's normal workflow when writing tests, enabling automatic creation of assertions based on information produced by actually running the program. The tool design was informed by an analysis of the assertions that developers write. It was validate with an analysis of how the assertions produced by this tool compare to those written by developers, and a user study about how developers use and think about a tool like AutoAssert.

# Preface

The work presented in this thesis was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey campus. All projects and associated methods were approved by the University of British Columbia's Research Ethics Board [certificate #H20-02151]. This material is the result of ongoing research at the Software Practices Laboratory.

I was the lead investigator, responsible for all major areas of concept formation, data collection and analysis, implementation, as well as manuscript composition.

Reid Holmes was the supervisory author on this project and was involved throughout the project in concept formation and manuscript composition.

Joanne Atlee was an advisor for this project and contributed to study design and manuscript edits.

Braxton Hall was a research assistant on the project, helping code the implementation behind the analysis in Chapter 2.

The material has not been published prior to this thesis.

# Table of Contents

<b>Abstract</b> . . . . .	<b>iii</b>
<b>Lay Summary</b> . . . . .	<b>iv</b>
<b>Preface</b> . . . . .	<b>v</b>
<b>Table of Contents</b> . . . . .	<b>vi</b>
<b>List of Tables</b> . . . . .	<b>viii</b>
<b>List of Figures</b> . . . . .	<b>ix</b>
<b>Acknowledgments</b> . . . . .	<b>xii</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
<b>2 Assertions in Practice</b> . . . . .	<b>4</b>
2.1 Methodology . . . . .	4
2.2 Results . . . . .	6
<b>3 Automatic Assertion Generation</b> . . . . .	<b>14</b>
3.1 Design methodology . . . . .	14
3.2 Tracing Program Values . . . . .	16
3.3 Identifying Assertion Categories . . . . .	17
3.4 Generating assertions . . . . .	18
3.5 Implementation . . . . .	19

<b>4</b>	<b>Evaluating Assertion Correctness . . . . .</b>	<b>21</b>
4.1	Methodology . . . . .	21
4.2	Results . . . . .	23
<b>5</b>	<b>User study . . . . .</b>	<b>25</b>
5.1	Methodology . . . . .	25
5.2	Results . . . . .	29
5.2.1	Pre-task survey . . . . .	29
5.2.2	AutoAssert tasks . . . . .	30
5.2.3	Post-task survey . . . . .	31
<b>6</b>	<b>Discussion . . . . .</b>	<b>35</b>
6.1	Improving Readability with Generated Assertions . . . . .	35
6.2	Limitations and Future Work . . . . .	36
6.3	Threats to validity . . . . .	38
6.4	Related Work . . . . .	39
<b>7</b>	<b>Conclusion . . . . .</b>	<b>41</b>
	<b>Bibliography . . . . .</b>	<b>42</b>
<b>A</b>	<b>Consent Forms and Surveys . . . . .</b>	<b>44</b>
<b>B</b>	<b>User Study Tasks . . . . .</b>	<b>53</b>
<b>C</b>	<b>Survey Results . . . . .</b>	<b>64</b>
C.1	Pre-task Survey Responses . . . . .	64
C.2	Post-task Survey Responses . . . . .	70
<b>D</b>	<b>Assertion Generation Details . . . . .</b>	<b>74</b>

# List of Tables

Table 2.1	Assertions encountered in practice. This shows the twelve high-level assertion categories categorized from 33,873 assertions in 18,937 tests from 84 projects. Note: some assertions can appear in multiple categories (e.g., <code>expect(val).to.exist.and.be.true</code> ) resulting in a total count larger than the number of assertions. Categories are based on assertion semantics and have been adjusted to account for semantic equivalents as described in Chapter 2.2	9
Table 2.2	Analysis of the 17,189 uses of the equality assertion methods to determine what category of assertions developers are actually using this operator for in practice. This shows that 22.5% of usages of the equality operator are asserting more semantically-specific behavior. . . . .	11
Table 4.1	Breakdown of the dynamic generation results. For assertion categories supported by AutoAssert, AutoAssert can almost always generate an assertion of the same category. . . . .	23



# List of Figures

Figure 2.1	Histogram showing the number of assertions per test case across 14,824 test cases with inline assertions. . . . .	6
Figure 2.2	Histogram showing the number of keywords per assertion across 20,418 assertions using the <code>expect</code> API . . . . .	7
Figure 3.1	Process used by AutoAssert to generate assertions. While a developer starts the assertion generation process for a specific variable in a test case, they also review the generated assertions to ensure both that the assertion operators are appropriate, and that the observed values match their expectations. . . . .	15
Figure 3.2	Usage of AutoAssert through IntelliJ context menu, adding an option to the context menu when a selection is made. . . . .	20
Figure 5.1	Usage of AutoAssert through the web interface created for the user study, replicating the context menu behaviour from the initial plugin version. . . . .	28
Figure 5.2	Three assertions generated by AutoAssert during a task. The first two were removed by the participant, the final most specific assertion was kept. . . . .	31
Figure 5.3	Breakdown of participant responses to the three main Likert-scale questions from the post-task survey. . . . .	32
Figure A.1	Consent form. . . . .	47
Figure A.2	Pre-survey. . . . .	50
Figure A.3	Post-survey. . . . .	52

Figure B.1	The tutorial task in the user study, before assertion generation. This test was created by me for the purpose of this study. . . .	54
Figure B.2	The tutorial task in the user study, after assertion generation. .	54
Figure B.3	Task 1 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock. . . . .	55
Figure B.4	Task 1 of 5 primary non-poison-pill tasks in the user study, after assertion generation. . . . .	56
Figure B.5	Task 2 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock. . . . .	57
Figure B.6	Task 2 of 5 primary non-poison-pill tasks in the user study, after assertion generation. . . . .	58
Figure B.7	Task 3 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock. . . . .	58
Figure B.8	Task 3 of 5 primary non-poison-pill tasks in the user study, after assertion generation. . . . .	59
Figure B.9	Task 4 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Typeset. . . . .	59
Figure B.10	Task 4 of 5 primary non-poison-pill tasks in the user study, after assertion generation. . . . .	60
Figure B.11	Task 5 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Typeset. . . . .	61
Figure B.12	Task 5 of 5 primary non-poison-pill tasks in the user study, after assertion generation. . . . .	62
Figure B.13	Poison-pill task 1 of 2 in the user study, before assertion gen- eration. This test was created by me for the purpose of this study. . . . .	62

Figure B.14	Poison-pill task 1 of 2 in the user study, after assertion generation. Here users should notice that the equality assertion is not a good fit for a result that should change every millisecond. . .	63
Figure B.15	Poison-pill task 2 of 2 in the user study, before assertion generation. This test was created by me for the purpose of this study. . . . .	63
Figure B.16	Poison-pill task 2 of 2 in the user study, after assertion generation. Here users should notice that the equality assertion is not a good fit for an identifier that should be unique every call via randomness. . . . .	63
Figure D.1	JavaScript code injected into projects to log the resulting value. Some types such as promises did not have unique implementations on the generation side at the time of writing. (1/2) . . .	77
Figure D.2	JavaScript code injected into projects to log the resulting value. Some types such as promises did not have unique implementations on the generation side at the time of writing. (2/2) . . .	78

# Acknowledgments

First I must thank Reid Holmes for his support, guidance, and input through every phase of this work, starting from encouraging me to undertake the Masters in the first place. He always made himself available to help whenever I needed it, and I left every meeting with my confidence higher than when it began.

I would like to thank Joanne Atlee for her guidance and expertise, and for her continued involvement even beyond the end of her sabbatical at UBC.

I would also like to thank Braxton Hall for all his contributions, even while working on grad school himself, as well as his friendship throughout my time at UBC.

My appreciation to the great community and individuals of the Software Practices Lab, I wish circumstances were different and I could have seen you all in person so much more.

I can't thank my family enough for all their support, both practical and personal, through both this degree and everything leading up to this point. I would never have made it this far without you.

And finally thanks to my partner Audrey Lu for her continued love, support, and encouragement through the highs and lows of completing this degree.

# Chapter 1

## Introduction

Automated testing continues to grow in importance in modern software systems. Unit tests [15] can provide quick feedback for developers, integration tests [12] can ensure components work in concert, smoke tests [3] provide rapid-high level feedback, and all play a central role in regression testing [7]. One commonality among all of these is that each test case needs assertion statements to verify the behavior of the code under test.

Assertions are short segments of code that play an out-sized role in test cases [16]. They are generally short, and are somewhat repetitive. But they are the arbiter of truth: a test case passes if all of its assertions pass; the test case fails if any of its assertions fail. A test case comprises code that invokes a behavior and a set of assertions that validate if that behavior is as expected. Assertions, like the test case itself, comprise code that requires effort to create and maintain. In this thesis I investigate how developers write assertions and develop a tool that can help them by automatically generating assertions for their test cases as they create them.

Specifically, I aim to improve the developer test-writing experience by automating the assertion-writing process with a human-in-the-loop tool called AutoAssert. AutoAssert is designed to fit into a developer's normal test-writing workflow, where test cases are frequently written alongside source code with developers asserting on invocations of the newly added code. To do this, AutoAssert uses dynamic analysis to observe the runtime value of the variable at the specified point in the program and determines the most appropriate assertions for the variable at

that program location, accounting for runtime variations that variables may exhibit (e.g., due to non-determinism).

The human-in-the-loop aspect of the tool is important: as the developer writes their test cases, they can select a variable instance of interest in the test case and generate local assertions about that variable; the developer is also involved in deciding which of the generated assertions to keep versus which to modify or remove. A generated assertion can also reveal to the developer unexpected values or properties of the variable of interest. The primary research question is whether the generated assertions are useful, despite the developer’s active role in reviewing and possibly editing the result.

A wide body of previous research has looked at ways to automate various aspects of test writing. Test generation approaches such as Randoop [9] and EvoSuite [5] can generate entire test suites (and their assertions); while these approaches could generate an individual test case, their primary use is to generate broad suites. In this thesis, I instead meet developers where they are and focus on supporting the developer who is writing specific individual test cases, by investigating how to ease the task of writing the test assertions.

Assertion generation approaches often leverage static analysis (e.g., UnitPlus [11] or Obsidian [1])). These approaches have the benefit of being fast, but developers need to verify that the generated assertions actually pass, as they are not based on actual runtime variable values and may fail when the test case is run. To counter this downside, dynamic approaches (e.g., Orstra [14], Eclat [8]) execute the code under test to ensure that the generated assertions are based on values the program produces when run. However these tools still lack a way of knowing the intentions of the tester and without this information these approaches may create assertions for any variable in the test file and after *every* time a variable is modified. While the tests thereby acquire test assertions about the variables’ runtime values, they also become bloated with assertions, degrading readability and the tests’ ability to serve as documentation for expected behavior. Bloat can be partially reduced by the use of helper methods for shared assertions, however this further reduces the understandability of the test.

Given the somewhat repetitive nature of assertions, machine learning approaches have also been applied to assertion generation. For example, the Atlas [13] system

uses a model trained on an assertion data set with hundreds of thousands of rows to generate new assertions when given the body of a test. As with other static approaches, Atlas cannot be certain its generated assertions are actually true and may face challenges generating more novel assertions. Since AutoAssert is dynamic and can observe the runtime value being tested, I focused my evaluation instead on determining if the type of assertion used was what developers might want, along with collecting feedback about how developers interacted with the tool and whether they found the generated assertions to be valuable.

**The primary goal of this research is to investigate the feasibility of generating test assertions and the utility of these assertions for developers.** To do this, I employed the following methodology: I first performed a quantitative analysis of developer-written assertions to understand what these look like in practice (Chapter 2). Using this understanding, I built a proof-of-concept tool called AutoAssert to generate the most commonly occurring assertions (Chapter 3). Using a quantitative simulation study I compared the assertions generated by AutoAssert to those written by developers (Chapter 4). Finally, I evaluated the utility of the AutoAssert tool in a formative evaluation with real developers (Chapter 5).

This thesis makes the following contributions:

- An empirical study characterizing the types of test assertions that developers write in real test cases.
- A prototype tool called AutoAssert which supports human-in-the-loop assertion generation.
- An empirical analysis 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.

## Chapter 2

# Assertions in Practice

I first seek to understand how developers write assertions in practice to inform the design of my assertion generation approach by investigating:

### **RQ1: What do test assertions look like in practice?**

This research question examines the scope, complexity, and semantic variation of developer-written assertions for real software systems. I will use the insight from this question to guide the design of my assertion generation approach so it can create the kinds of assertions that developers write in practice. The study could also help others who seek to better understand developer-written assertions.

### **2.1 Methodology**

To answer **RQ1**, I performed a quantitative study by statically analyzing the developer-written test assertions present in open-source projects. I selected these projects by examining all JavaScript and TypeScript projects published on npm<sup>1</sup> that are dependent on the Chai assertion library<sup>2</sup> and Mocha test framework<sup>3</sup>. Mocha and Chai are similar to other test frameworks, including those in other languages such as JUnit. From these projects, I was interested in larger, maintained software systems closer to those that would be found in industry (i.e. avoiding personal

---

<sup>1</sup><https://npmjs.com>

<sup>2</sup><https://www.chaijs.com>

<sup>3</sup><https://mochajs.org>



projects). As such I selected only those projects that link to open-source repositories on GitHub that have at least 100 stars. From these 105 open source projects, I analyzed all 33,873 Chai test assertions from all 18,937 test cases written using the Mocha test framework. For my test counts, I define a test case as any method call to the Mocha framework test creation method, called `it`.

I created a static analysis tool, based on the TypeScript parser for TypeScript and JavaScript source code, that parses the Mocha test files in each project's repository and looks for AST elements that correspond to calls to the Chai `expect` and `assert` functions, which are the core assertion statements used by the Chai assertion library. The parser extracts three pieces of information from each non-trivial test assertion<sup>4</sup>:

1. *The element under test.* For my data set of 33,650 non-trivial assertions, this was most commonly either a property access on an identifier (35.8%), an identifier referring to the result of a method call or complex expression (31.3%), or an inline method call (18.1%).
2. *The specific assertion method.* This is a method from the Chai API, such as 'equals', 'length', or 'null' that imparts specific constraints on the variable under test.
3. *The expected value to which the variable under test is compared.* In my data set, this is most commonly a literal value such as a string or a number (61.6%). In 24% of the cases, the expected value is embedded in the name of the Chai method (e.g., `expect(val).to.be.true`, `expect(val).to.be.null`). In the remaining cases, the expected values were identifiers (12.3%), property accesses (6%), complex expressions (3.4%), or method calls (2.7%). A single assertion may compare the variable under test with more than one expected value, thus the percentages of the expected-value categories add up to more than 100%.

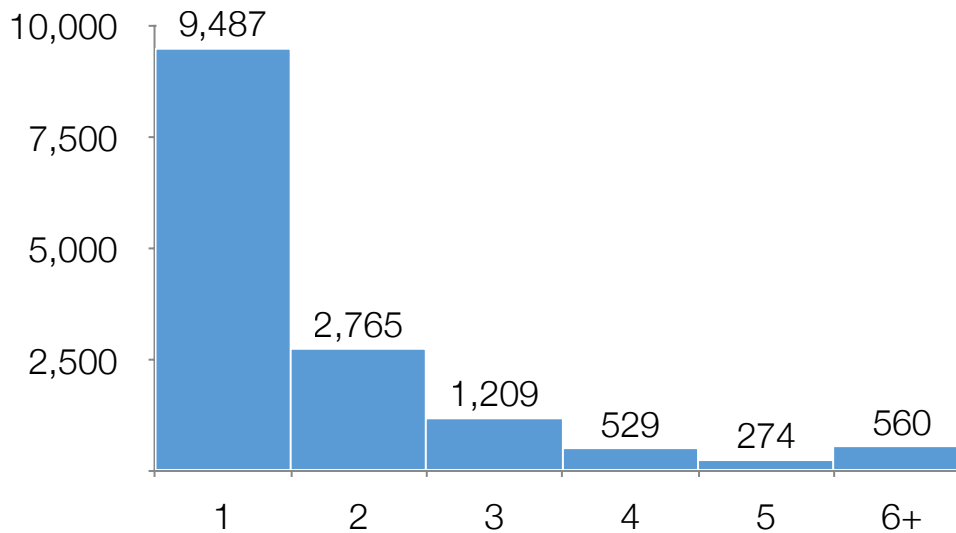
---

<sup>4</sup>Where a non-trivial test assertion is one that refers to program variables, as opposed to assertions like `expect.fail()`. Such trivial assertions are control flow checks (or may potentially be API misuse), but not interact with program elements.

## 2.2 Results

I next analyzed this set of assertions to learn how many assertions are used per test, how complex they are, what kinds of assertions developers use, and how they construct them.

*Assertion Density* Developers add assertions to their test cases to validate program behaviours; in practice I found that most test cases contain relatively few assertions. Of the 18,937 test cases in my data set, 4,113 test cases (22%) contain no test assertions<sup>5</sup> or rely on helper functions to call the assertions (making it difficult to attribute the assertion to the calling code).



**Figure 2.1:** Histogram showing the number of assertions per test case across 14,824 test cases with inline assertions.

Figure 2.1 shows the distribution of the number of assertions per test case among the 14,824 tests that contain at least one test assertion: the test cases have a mean of 1.4 test assertions per test and a median of 1 assertion per test; 17.4%

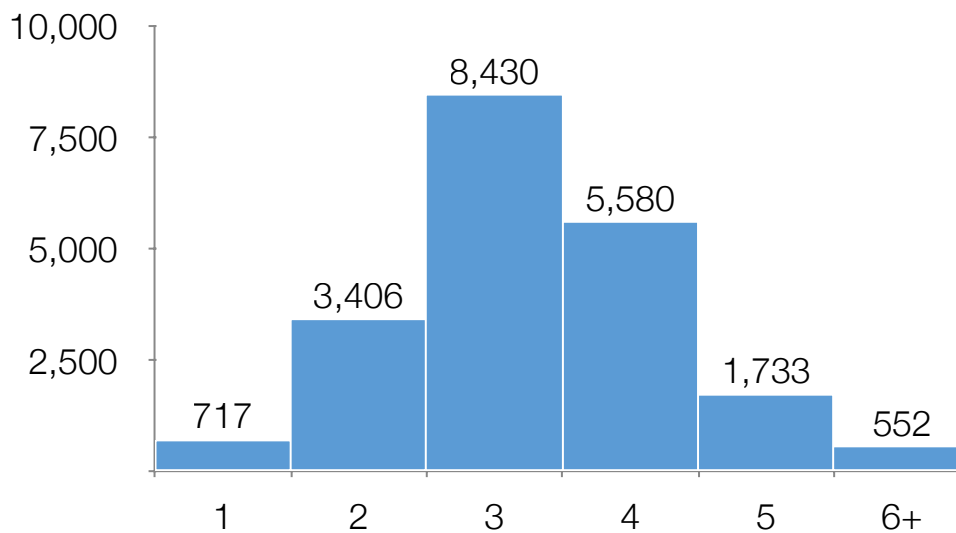
---

<sup>5</sup>Some assertion-free tests simply check to see if the code under test throws an error, which implicitly causes the test to fail. Alternatively they may serve as entry points for running the code to debug or view output.

of the test cases have three or more assertions; and the outlying test case with the most assertions had 183 assertions.



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



**Figure 2.2:** Histogram showing the number of keywords per assertion across 20,418 assertions using the `expect` API

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

```
# Assertion Statement
2 expect(save) .throws(err)
3 expect(conf) .to.equal('ICSE')
4 expect(reviewers) .to.not.include('r2')
11 expect(pages) .to.exist.and.
    to.be.at.least(10) .and.at.most(12)
```

Figure 2.2 shows the assertion complexity across the 20,418 assertions using the `expect` assertion library I analyzed. Most assertions 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.

**Table 2.1:** Assertions encountered in practice. This shows the twelve high-level assertion categories categorized from 33,873 assertions in 18,937 tests from 84 projects. Note: some assertions can appear in multiple categories (e.g., `expect(val).to.exist.and.be.true`) resulting in a total count larger than the number of assertions. Categories are based on assertion semantics and have been adjusted to account for semantic equivalents as described in Chapter 2.2

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

*Assertion Categories* To understand the kinds of assertions developers were writing, I categorized all of the assertions in my data set. I grouped assertions by their keywords as described in Chapter 2.2, discarding keywords that were only used 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, I used the keyword name and the official documentation from the assertion library to confirm the kind of behaviours validated by each assertion keyword. I then performed an open card sort [2] 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`. Specific behaviours may be different between these methods, however they all operate on the same details of the value under test. This resulted in twelve categories; these are shown in Table 2.1. Each category corresponds to a kind of semantic check developers have expressed in their test cases to validate behavioural correctness in the code under test.



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

*Assertion Equivalence* There are multiple semantically equivalent ways for developers to write any given assertion. Forms of equivalence generally fall into two categories. In the first, the assertion library provides multiple 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 when structuring their assertions. For instance, writing `expect(array.length).to.equal(1)` instead of `expect(array).to.have.length(1)`. This kind of equivalence, with one option being a specific assertion operator and the other being framed as an equality check, was by far the most common semantic equivalence I observed.

Before I could run my analysis of assertion equivalents, I needed to decide what equivalents developers may write so that they can be set as targets for my static analysis. To do so I first clustered the assertions in my data set into categories.

Then from each of these categories I took a random sample of 5% of each categories assertions. While a random sample of the entire dataset would likely achieve a similar breakdown, this lowers the odds of one category of behaviour being under represented. I then read through all the sampled assertions and noted down any instances where I observed assertions that were semantically equal but stylistically different, both within or between assertion categories. Some behaviours that were not present in the sample, but logically followed from behaviours seen in the sample, were also added. For instance, checks on `result.length` for arrays were present in the sample, however checks on `result.size()` for sets were not. One notable exception to this was numeric operators, such as greater than (`>`) or less than (`<`). To my surprise, despite these operations being present through provided assertions methods such as `above` and `below` there were no usages of the equivalent operators in my sample. As it was a clear opportunity for semantic equivalence, I decided to include this category despite its absence from my sample.

**Table 2.2:** Analysis of the 17,189 uses of the equality assertion methods to determine what category of assertions developers are actually using this operator for in practice. This shows that 22.5% of usages of the equality operator are asserting more semantically-specific behavior.

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

I examined the form of equivalence where developers use equality in place of more semantically rich assertions in more detail, as my preliminary analysis of assertion categories found that more than 50% of all assertions were checks on equality. Of these, I found that at least 22.5% of the developers' uses of equality-based assertion operators could be written with a more specific assertion operator (see Table 2.2).

There are two main advantages to using specific assertion operators rather than framing them as an equality-based assertion (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:

```
Ex1. expect(arr).toContain('c')
```

```
Ex2. expect(arr.indexOf('c') >= 0)
     .toEqual(true)
```

The second advantage is that when an assertion fails, if the assertion is more specific, then its error message can be made more descriptive for the developer. Consider the failure messages that correspond to the two previous assertions; the more specific assertion is able to provide a more specific and meaningful error message:

```
Ex1. AssertionError: expected
     [ 'a', 'b' ] to include 'c'
```

```
Ex2. AssertionError: expected
     false to equal true
```

The results of this analysis suggest that assertion generation tools should consider accommodating possible assertion equivalents and should carefully consider generating the more specific assertion form.



Developers may write assertions for the same behaviour in a variety of ways; assertion generation approaches should be aware of these differences, but also encourage consistency.



#### **RQ1 Summary**

Most test cases contain relatively few assertions with a median of 1 assertion per test. Assertions are generally simple, typically with enough tokens for a single check and possibly a negation. Longer assertions that chain multiple checks together in one statement are rare. While half of assertions are based on equality, the rest have a wider variety such as checking values types, sizes, ranges, and patterns. Semantically equivalent assertions are common, especially with respect to checking equality; developers often choose the equality based assertion methods and then use the arguments to express more specific semantics.

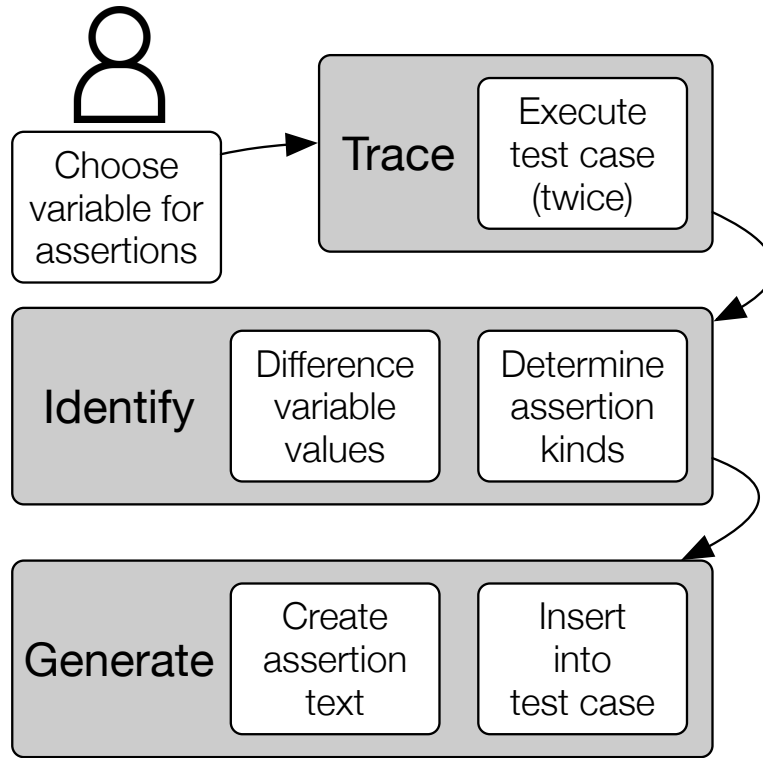
## Chapter 3

# Automatic Assertion Generation

Using the information I learned about the nature of developer-written assertions in Chapter 2, I created a prototype tool, called AutoAssert, that automatically generates many common kinds of assertions. Generating assertions proceeds in three main phases, as depicted in Figure 3.1. First, the developer selects a variable in their test case for which they wish to have assertions generated; the test case is executed (twice) and a program trace recording the values assigned to the variable are collected. Second, appropriate kinds of assertions are selected based on the values assigned to the variable at runtime. Finally, the assertion text is generated and inserted into the test case for the developer to review. Each of these steps is described below.

### 3.1 Design methodology

When designing AutoAssert I examined the different categories in Table 2.1 and noted that many categories would be straightforward to generate, while others would be much more complicated. With a dynamic approach, the most directly generateable assertions are from the `Equality`, `Boolean`, `Length`, `Existence`, and `Type` categories. These categories each have clearly observable properties that imply exact expected values they should be compared against. One more opaque category that AutoAssert is able to support, is `Throw`. The convention, at least for the Chai API, is to wrap the function call you wish to test in an any-



**Figure 3.1:** Process used by AutoAssert to generate assertions. While a developer starts the assertion generation process for a specific variable in a test case, they also review the generated assertions to ensure both that the assertion operators are appropriate, and that the observed values match their expectations.

mous function with no arguments. That function is then passed to the assertion which invokes it. As such, if the value under test is a function with no arguments, I can invoke it to determine if it throws or not and generate an assertion accordingly. A list of all the assertions that AutoAssert generates can be found in Listing D.1.

Some 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 element under test is the important one to assert on and can not 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. While the `Calls` category could be compared against whether a function has been called at all, how many times, or with what arguments, it is not clear whether these are often needed. Finally, while easy to implement, `Truthiness` was the least seen category and I considered it to be largely redundant given checks for `Boolean` and `Existence`, and its use is also discouraged in the Chai API documentation. As such it was intentionally excluded to avoid bloating the generated assertions. Examples of these unsupported assertions are provided in Listing D.5.

## 3.2 Tracing Program Values

While a variety of approaches could be used to determine the values of variables under test, AutoAssert takes a dynamic approach of executing individual test cases and recording the values assigned to variables at runtime.

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

**Listing 3.1:** An example test case. A developer would generate assertions by selecting `uuid` and invoking AutoAssert.

For example, for the test case in Listing 3.1, the developer can right click on the `uuid` variable and invoke AutoAssert. The tool invisibly injects tracing code into the individual test case, immediately after the statement containing the variable instance selected; executes the test case twice; and records the values assigned to `uuid`. After the test case is executed, the tracing code is removed.<sup>1</sup>

Listing 3.2 shows the recorded values of variable `uuid` for the two execu-

---

<sup>1</sup>For TypeScript projects, the project is incrementally compiled after injecting the tracing code, but before the test is run.

tions. Running each test case twice<sup>2</sup> 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). Knowing which properties of a variable change between executions decreases the probability that AutoAssert will generate assertions that are too strong. In this case, I can see that the runtime values of `uuid` are strings in both test-case executions and have the same length but different values. If the 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), 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<sup>3</sup>.

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

**Listing 3.2:** 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. Further examples regarding results from running the test twice can be found in Appendix D.

### 3.3 Identifying Assertion Categories

Once a variable has been traced and recorded, the next step is to identify which kinds of assertions are applicable given the variable’s runtime values. My main goal here is to identify several categories of assertion that make sense for the invari-

---

<sup>2</sup>I hypothesized that non-deterministic tests will either be fully random, or have their potential differences determined by external factors such as time, platform, or environment variables. Under this assumption running more than twice would be unlikely to produce any new variations, and this chance would not be worth the increase in runtime

<sup>3</sup>Note that JSON’s standard serialization of objects (`stringify()`) does not serialize method names.

ant portion of the recorded value and order these categories in increasing strength. While this often results in ‘extra’ assertions, the additional assertions increase the specificity of the resulting error messages. For example Listing 3.3 shows three assertions for a simple object<sup>4</sup>. 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. Developers could also easily delete any assertion they deem redundant or too strict to suit their personal preference.

```
cont val = db.getPaper("thesis");
expect(val).to.exist;
expect(val).to.be.an("object");
expect(val).to.equal({ author: "Lucas", year: "2020" });
```

**Listing 3.3:** An example set of Chai assertions generated after a line of test code that returns a simple object.

If the developer selects a variable of interest that does not have a clear value such as a function, then only existence and type checks (and no equality check) are generated. Wherever possible, the more specific assertion categories from Table 2.1 are used over equality to ensure the resulting error messages are as specific as possible. With respect to equality checks, strict-value equality checks are generated for primitives whereas objects and arrays are checked for deep equality<sup>5</sup>. The outcome of this step is a list (ordered by increasing strength) of assertion categories that make sense for the object’s values observed for two execution traces.

### 3.4 Generating assertions

The process of generating the assertion text is straightforward once the assertion categories and order of generated assertions are determined. The generated text is a

---

<sup>4</sup>Additional examples of generated assertions can be seen in Appendix B.

<sup>5</sup>Deep equality checks values of nested properties and elements, instead of memory address equality.

multi-line string that is injected into the developer’s test code immediately following the program statement that contains the variable instance on which AutoAssert was invoked. To help developers understand the rationale for the generated assertions, a short inline comment can be optionally appended to the end of each assertion explaining what behaviour the assertion is checking. Since AutoAssert is designed to be a human-in-the-loop system, I expect developers will do their due diligence in reviewing generated code, accepting (leaving unchanged), rejecting (removing), or modifying assertions.

### 3.5 Implementation

My initial implementation of AutoAssert supports JavaScript and TypeScript. Any language that supports dynamic variable introspection is amenable to this approach however. These languages are well suited to the generation of test assertions as inspecting the values of objects is relatively straightforward. The JavaScript ecosystem supports a large number of assertion libraries; I opted to support the Chai assertion library for its clean syntax variety of assertion options, although any library could have been chosen.

To embed AutoAssert into a developer’s normal workflow, it was implemented as an extension to the IntelliJ IDEA Ultimate<sup>6</sup> because of its strong support for extensions, its wide user-base, and its support for both JavaScript and TypeScript. Figure 3.2 shows what AutoAssert looks like within IntelliJ. A full demonstration of AutoAssert in use is available online<sup>7</sup>.

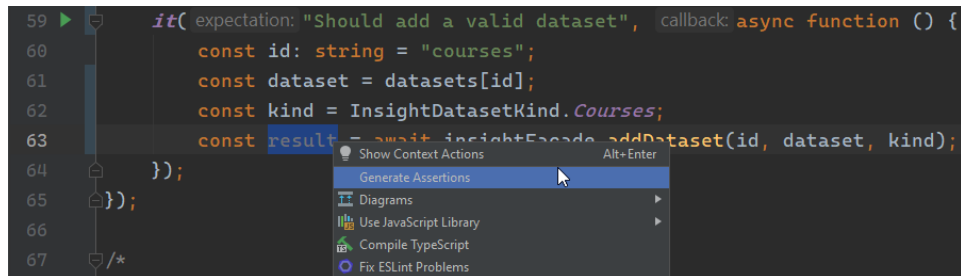
The user interface is a simple context menu option “Generate Assertions” that appears whenever a developer right-clicks on a variable in a test case in the IDE. The plugin then creates a run configuration to execute only the test that the target variable is found in<sup>8</sup>. It then adds a before run task to invisibly inject the logging code, and triggers the run configuration to execute. As IntelliJ does not to my knowledge provide a way to get a callback once a run completes, AutoAssert

---

<sup>6</sup><https://www.jetbrains.com/idea/>

<sup>7</sup><https://youtu.be/w1MoeZxfJko>

<sup>8</sup>This is to make the execution timely, however it does mean that AutoAssert may not work correctly for tests that are not proper unit tests, and require state set up by prior tests. This is a design decision for the plugin however, not a requirement of the approach more broadly



**Figure 3.2:** Usage of AutoAssert through IntelliJ context menu, adding an option to the context menu when a selection is made.

instead watches the output directory for the expected logging file. As soon as the file appears, the original file without the logging code is restored and assertions are inserted. The time required for this process depends on the how long it takes to run the test case itself through IntelliJ; the tool itself has no meaningful overhead. The most computationally expensive steps of running AutoAssert itself should be the disk operations associated with reading and writing the logging file, though no thorough performance analysis was performed.



## Chapter 4

# Evaluating Assertion Correctness

To evaluate the quality of the assertions generated by the prototype AutoAssert tool, I performed an empirical simulation comparing my generated assertions against those included in real software projects.

**RQ2: Can my approach 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 a developer did for the same input value. Being able to consistently produce assertions in the same category developers want is crucial for making an assertion generation tool that developers will use and trust.

### 4.1 Methodology

To perform my empirical simulation, I selected 10 open source projects with developer-written programmatically-executable test suites. Projects were selected using my final list of candidate projects from Chapter 2.1, sorted by stars. I then chose the first 10 projects whose test suites I was able to programmatically execute. I statically analyzed each project to find all developer-written assertions in the project<sup>1</sup>. This resulted in 1,335 assertions across the 10 projects. For each of these assertions I extracted the element under test, the left-hand side of the assertion, and used this as input to trigger assertion generation with AutoAssert. For each of these, Au-

---

<sup>1</sup>Each test was executed and those that failed were excluded.

toAssert generated one to four assertions.

After the assertions were generated they were compared to those written by developers. I considered two assertions to be equivalent if they belonged to the same category (accounting for equivalent forms as described in Chapter 2.2). This evaluation did not try to compare the exact assertion text, as that would require either using an exact text match as the metric for success, or knowledge of the code under test to judge whether the generated assertion was equivalent to the original assertion.

Listing 4.1 shows an instance of a category match. In this case one of the generated assertions happens to be identical, but this would also be considered a match if any other equality-based assertion method were used. Listing 4.2 shows a category miss, where the original assertion was intending to do a type based check, however my assertion checked that the value was going to be exactly `null` as this was the value error actually had assigned to it when the test executed.

```
// 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 4.1:** 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 4.2:** Example of generated assertions that did not contain the same category (type) as the original assertion. This is considered a miss.

**Table 4.1:** Breakdown of the dynamic generation results. For assertion categories supported by AutoAssert, AutoAssert can almost always generate an assertion of the same category.

Category	Count	% of Sample	% Hit
Equality	804	60.22%	99.00%
Boolean	101	7.57%	100.00%
Existence	101	7.57%	100.00%
Type	65	4.87%	98.46%
Length	59	4.42%	96.61%
Throw	9	0.67%	100.00%
Numeric	118	8.84%	0.00%
Truthiness	58	4.34%	0.00%
Properties	10	0.75%	0.00%
Inclusion	8	0.60%	0.00%
Calls	1	0.07%	0.00%
Patterns	1	0.07%	0.00%
<b>Total</b>	<b>1335</b>	<b>100.00%</b>	<b>84.49%</b>

## 4.2 Results

For the assertion types AutoAssert supports, if the developer wrote an assertion of that kind, AutoAssert generated an assertion of the same kind in at least 96% of cases. For categories AutoAssert does not support, no equivalent assertion could be generated. Since AutoAssert supports many of the most commonly-used kinds of assertions (Table 4.1), the tool was able to generate assertions in the correct category 84.5% of the time.

One of the challenging scenarios for correct generation is when developers make assertions about what a value is *not*. Listing 4.2 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, as the space of what a value is *not* is infinitely large. To understand the prevalence of negated assertions, I checked my results for the presence of the `not` modifier or methods such as `notEquals`. Of successful generations of equality assertions, only in 17 cases did the original equality assertions contained a `not`. There were also 23 existence assertions with

`not` modifiers, however AutoAssert is able to generate these correctly so this is not a concern for this category. No other category had more than one `not` in the original assertion. Another failure mode was assertions where the element under test is equal to a certain named function, such as:

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

This would require being able to identify the name of the function returned and the proper scoping to reference it in an assertion. One additional detail that could complicate assertion generation is when the value under test may be different on different test runs. In my sample 40 tests (3%) had elements under test that were different when the test was run a second time.

While this evaluation showed that for most assertions, AutoAssert could generate assertions of the appropriate category, it did not check for exact matches of the assertion value, although since this is observed at runtime it is guaranteed to *pass*, if not be identical to the assertion the developer wrote themselves. This consistency is important because in the context of a developer invoked tool, if a developer lacks confidence that the tool can produce the type of assertion they want, they may forego using the tool altogether. It also reaffirms the concept that a reasonable portion of assertions can be generated by AutoAssert.

#### 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 96% to 100% of the time in supported categories, for an overall accuracy of 84% when considering all categories. The most challenging scenarios for this generation approach in assertion categories I target were those involving saved references to functions, and assertions on what a value is not.

## Chapter 5

# User study

With AutoAssert able to generate appropriate assertions for 84% of cases, I next performed a formative user study with developers to learn how they perceive the assertions generated by AutoAssert. Specifically, I 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?**

This research investigates whether the assertions generated by AutoAssert are considered valuable by real developers. While there is a wide range of possible assertion generation tools and kinds of assertions that could be generated for developers, I hope that examining the strengths and weakness of AutoAssert will help guide the development of future assertion generation tooling.

### 5.1 Methodology

Participants for my user study were recruited primarily through various software development related forums on Reddit<sup>1</sup>. 23 individuals, 21 men and 2 women, participated in the study in some capacity, with 17 completing all the 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

---

<sup>1</sup><https://www.reddit.com/>

participants then used a web-based version of AutoAssert to generate assertions for eight test cases, and concluded with a post-task survey. This online format was not my preferred modality, but in response to COVID-19 I decided this was the prudent way to move forward and engage engineers.

*Pre-task survey* The pre-task survey collected participant consent, gathered relevant demographic information along with years of experience programming professionally, 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:

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

The pre-task survey materials can be viewed in full in Appendix A. At the conclusion of the pre-survey, participants were automatically forwarded to my browser-based version of AutoAssert.

*AutoAssert tasks* To answer **RQ4** I sought to have participants use AutoAssert and evaluate the assertions it generated 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 the code editor<sup>2</sup> hosted in the browser instead of an IDE. The UI for the tool was otherwise the same: participants could view the test and associated source file, invoke AutoAssert with a context menu by clicking on the variables they wanted assertions for, and these assertions would be injected into the browser-based editor for them to manipulate as they saw fit. The UI for the browser-based AutoAssert is shown in Figure 5.1.<sup>3</sup>

---

<sup>2</sup>Based on Ace <https://ace.c9.io>

<sup>3</sup>The browser-based AutoAssert implementation and all experimental tasks can be found at <https://se.cs.ubc.ca/AutoAssert/index.html>.

The task component of the study has participants use AutoAssert to generate assertions for a series of test cases. Once they have generated assertions, they can delete, modify, or add any assertions they want until they feel they are appropriate for the test. Participants can also leave any comments they want in the code itself if there is anything they would like to note about the assertions. There is one training task and seven experimental tasks that participants must complete. The training task was a very simple test case along with comments that showed the participant how to invoke AutoAssert and guided them through a sample task. Additionally instructions were also always present in a header at the top of the page.

Five of the seven experimental tasks were taken from two open source projects: Typeset<sup>4</sup> is a string manipulation library for replacing ASCII characters with more visually appealing Unicode characters. Nock<sup>5</sup> 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 them to concentrate on understanding the test code instead of the product code, I 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 tests from the same project always appear alongside each other so users do not need to recall what a prior project was about.

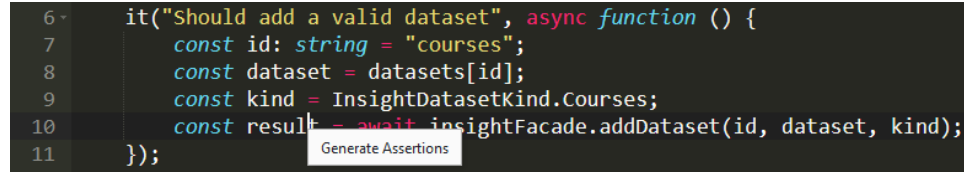
I also included two ‘poison pill’ tasks that appeared in between the two randomized blocks of tasks. These tasks generated assertions that were intentionally poor to evaluate how developers reacted to poor automation. Specifically, these tests generated strict equality checks on code that generated unique results on every run (running twice to check for differences was purposefully disabled here). For this task I want to make sure that participants modified the generated assertions or commented on the challenges in the post-survey. This ensures that they are 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 after each generation as well as when they 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. All tasks as well as the assertions generated when users interact

---

<sup>4</sup><https://github.com/davidmerfield/Typeset>

<sup>5</sup><https://github.com/nock/nock>

with them are provided in Appendix B.



```
6 it("Should add a valid dataset", async function () {
7   const id: string = "courses";
8   const dataset = datasets[id];
9   const kind = InsightDatasetKind.Courses;
10  const result = await insightFacade.addDataset(id, dataset, kind);
11  });
```

Generate Assertions

**Figure 5.1:** Usage of AutoAssert through the web interface created for the user study, replicating the context menu behaviour from the initial plugin version.

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

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

The first three questions were based on a Likert scale while the rest were free-response. The post-task survey materials can also be viewed in full in Appendix A.

These questions seek to gain further insight into **RQ4** in addition to data recorded about how developers performed the AutoAssert tasks previously.



## 5.2 Results

In this section I discuss the results from my user study, including the pre-survey about test development behavior, behavior during the assertion generation tasks, and post-survey feedback about AutoAssert.

### 5.2.1 Pre-task survey

23 participants completed my pre-survey (91% male). 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. 17 participants completed the AutoAssert tasks and post-survey in full.

**Assertion writing process.** Two participants (9%) stated they did not typically include assertions in their tests, beyond the tests' built in ability to detect thrown errors. 11 (48%) included assertions as post-conditions while 10 (43%) use assertions as both pre- and post-conditions in their test cases.

Participants employed a wide variety of processes when creating their test cases and assertions. While for simple unit tests developers may have some assertions in mind when starting their test case, for more complex test cases one participant developers mentioned that, *"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, while most participants (91%) mentioned they often knew which assertions they would write in advance of creating a test case, a surprising number (65%) mentioned running the code and inspecting its output in one form or other. Taken together, these comments fit well with the process AutoAssert embodies as it suggests these developers often have (at least initial) implementations before writing their test cases.

**Existing tools.** 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. 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 their code under test. While creating assertions they frequently run their test and inspect the values returned by the code under test while creating and debugging their assertions, even if they had an idea of what the result should be ahead of time. Assertions were often used as both pre-condition and post-condition checks in test code.

### 5.2.2 AutoAssert tasks

17 participants completed all AutoAssert tasks. 14 of the 17 noted that ‘poison pill’ tasks had problems (either by changing the tests or making inline comments), suggesting a high level of engagement with the AutoAssert output; I exclude the three developers who did not notice these problems below (for a total of 14 participants).

Five participants left the assertions generated by AutoAssert in place for the five good tasks, suggesting they agreed the assertions generated by AutoAssert were appropriate. For the nine participants who changed the assertions, the most common behavior (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 tests involving server responses noting they would create assertions for both the response code and body instead of one or the other. This reflected the structure of the task (setup to evaluate the response code but not body) rather than 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 that appeared in both the test setup and generated assertion into a variable for cleaner code. Three participants made changes to the expected values that did not match the program behavior in at least one test, presumably misunderstanding the intended behavior of the code under test.

**AutoAssert task summary.** Ultimately, the 14 engaged participants generated

```

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);
});

```

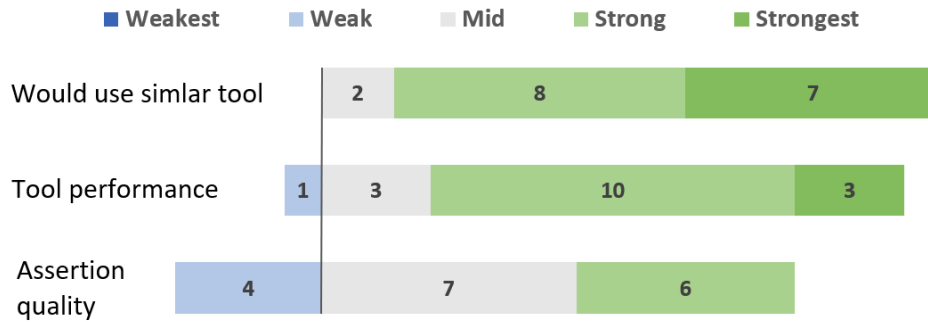
**Figure 5.2:** Three assertions generated by AutoAssert during a task. The first two were removed by the participant, the final most specific assertion was kept.

a combined 210 total assertions using AutoAssert for the 75 non-poison-pill tasks they performed. Of these assertions, they “approved” that 174 (83%) were correct and “rejected” 36 (17%) by removing or modifying them. Figure 5.2 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. 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 (94%) of respondents found the tool performance to be acceptable or better. Naturally, I was most interested in how developers perceived the quality of the assertions. Only 4 (24%) of participants found AutoAssert’s automatically-generated assertions to be *somewhat worse* than those they would write themselves. More details about these responses can be seen

in Figure 5.3.



**Figure 5.3:** Breakdown of participant responses to the three main Likert-scale questions from the post-task survey.

**AutoAssert assertion quality.** As previously mentioned, only 4 participants found AutoAssert’s assertions worse than they would have written in their own test cases. Participants who felt that the assertions could be improved were split in opposite viewpoints as to what ‘better’ means for assertion quality. Some participants suggested that as earlier less strict assertions were functionally captured by the later more strict assertion, they were unnecessary and should be removed. Other participants appreciated the increased descriptiveness of the less strict assertions, and worried that the most strict assertions might be overly specific and preferred to remove them. 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 to account for these two philosophical viewpoints might be warranted.

Participants did note that AutoAssert did produce 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 my self.*” and “*They were very thorough, which encouraged me to include more of them.*”

Unsurprisingly, many participants commented on the poison-pill tasks.<sup>6</sup> These participants noted that assertion generation would not be appropriate for cases with randomness or change. Fortunately, the real AutoAssert tool does handle these situations (in at least a basic way) that would not have failed for these two tasks if it were enabled. Participants also expressed concern about other scenarios not included in the tasks. JavaScript allows for return values of any type, which one user noted would invalidate type checks. Another participant noted that it would be hard to handle cases where only a substring of a result (part of the Inclusion category) was the only important part of the return value. Multiple participants noted the fact that deep equality checks would be decreasingly appropriate as objects or arrays under test became larger.

**Improving assertion generation.** Many participants had suggestions for improving assertion generation tools in general. For example, having an explicit configuration option for adjusting the strictness of the generated assertion, suggesting having the 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 code base’s coding style and conventions (e.g., for quotes, line lengths, and indentation). One participant suggested starting from the code under test rather than the test case: they wanted to identify values in the code that they would like assertions to be generated for in the test case. One participant, rightly, was worried that automated assertion generation “*may offer the developer a false sense of security*”. I fully agree: this is why I think it is important that these systems be human-in-the-loop so 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 behavior, not create passing assertions for incorrect behavior).

---

<sup>6</sup>Note: Participants did not know that these were intentionally ‘poor’ tasks.

#### **RQ4 Summary**

Developers found the assertions created by AutoAssert to be broadly useful at validating program behavior for the test cases they evaluated, with only 4 of 17 saying the assertions produced were worse than what they would write. Additionally 15 of 17 developers noted they would be interested in using a tool like AutoAssert in their development environment. Preferences were mixed among those who suggested improvements, with some appreciating the inclusion of less strict assertions and concerned about over-fitting, and others finding the less strict assertions superfluous. I hypothesize that additional personalization (e.g., for strictness and style) could address the majority of concerns developers identified with the generated assertions.

## Chapter 6

# Discussion

Here I describe some implications for future assertion generation approaches, future work for AutoAssert, and threats to validity. In this thesis I have confirmed that most automated test cases have few assertions and those assertions do not tend to be too complicated (Chapter 2). I have also found that Assertion generation can fit within developer testing workflows and that they respond positively to generated assertions (Chapter 5). There is also a clear space for improving personalization and configuration options for generated assertions, along with balancing the strictness of a behavioral check with the brittleness of that check over time.

### 6.1 Improving Readability with Generated Assertions

While not the focus of this study, reading through large numbers of sampled and generated assertions lead to observations about how using assertion generation tools could improve test readability. One problem observed in my 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, readability suffers. Generated assertions would not make this mistake, and would further improve readability through consistency. While there are many different assertions methods and semantically equivalent forms for the same functionality, generated assertions

would be consistent in their selection of assertion for a given behaviour. Additionally when multiple assertions are produced they will always be produced in the same order. When assertions are consistently produced in this fashion it is much easier to understand the expected behaviour at a glance. Much like clean code is desirable to allow for easier understanding and maintenance, clean assertions can have similar benefits for test code.

## 6.2 Limitations and Future Work

I know of a few areas where the heuristic approach for generating assertions and implementation of AutoAssert could be improved.

**Adapting to project style** Given that developers can write assertions in many different ways, AutoAssert should try to increase the probability that the generated assertions will match developer expectations by choosing the equivalent assertion form that best matches those used by a given project. For example, even a simple assertion to check for `null` can be written three different ways (Listing 6.1). While the first of these is the most specific and would be preferred in the absence of any other input, an option should be provided to allow the developer's project to be statically scanned to see if they have a preference among different equivalent assertions for any assertion category.

The tool would statically analyze all of the test cases in a project to determine the most common equivalent form for different kinds of assertions. This process would only need to be done once, but could be re-run at any time to get up to date data. Selecting project-specific equivalent forms would not be required; if the static analysis has not been run on the project (or there are not yet any test cases in the project), AutoAssert would select the most common form that I identified in my initial empirical study (Chapter 2.2). Additional user options could allow for customization features such as producing only the strongest assertion possible for the test and suppressing the more verbose checks, or user created equivalent forms.

**Properties within complex objects.** One common usage scenario for assertions is to make a method call that returns a complex object, then the developer makes an assertion on a specific property in that object. While a developer could choose to instead save that specific property in a variable and then assert, it would



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

**Listing 6.1:** Semantically equivalent assertions for checking whether `value` is `null`.

be more natural for their workflow if a tool could suggest or determine the correct property. One extension I aim to examine is adding interactive support allowing users to explore and choose which property to assert on for languages with static class definitions. This would improve the selection of assertion targets and the specificity of the assertion methods.

**Support additional 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 inferable from observed values as they could require isolating a particular sub-element 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. [13]. Additionally project-specific static analysis could be used to identify what assertion methods are being used for a given data type or API name. If other assertions predominantly use inclusion or range methods, the tool could pivot to suggesting those kinds of assertions.

**Updating assertions in failed tests.** Maintaining test suites requires considerable developer effort [6]. AutoAssert could help developers update their tests to account for new dynamic behavior by allowing them to automatically update the value of existing failed assertions in their test suites when assertions start failing and need to be updated. While this is not currently supported, the current tool could be adapted to support this use case.

### 6.3 Threats to validity

My experimental design leads to several threats to validity that should be considered.

**External validity.** The nature of the assertions developers write may be influenced by my choice of language and framework and my focus on JavaScript/TypeScript may not generalize. The same can be said for the projects I selected my AutoAssert user study test cases from and performed my quantitative study on: these projects and tasks may not be representative of all test suites and test cases. I expect that the strongest impact of language and framework choice would be on the breakdown of categories shown in Table 2.1. For example I 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 I do not believe language or library selection weakens the rest of the work in other contexts, and that the motivations behind developer-in-the-loop dynamic assertion generation are still broadly applicable. AutoAssert’s generated assertions are also molded by the language I chose to support. JavaScript is well suited to creating and comparing literal values, and concepts like member visibility (e.g., private, protected) cannot hide data from being evaluated by assertions. Prior work has shown how these limitations can be overcome: Tao Xie’s work [14] has shown how 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.

**Internal validity.** The AutoAssert tasks my participants completed were based on projects and tests that they were not familiar with; this is unusual as test-writers are typically familiar with the code they are validating. As such, participants may be more deferential to automated tools than they might otherwise be. Since the user study focus was on the assertions themselves rather than the test outcome, participants might have behaved differently if they were running the test cases in their own familiar environments; unfortunately, pivoting online was a consequence of COVID-19 I felt had to be made.

## 6.4 Related Work

A variety of past approaches have investigated generating assertions using static, dynamic, and machine-learning based approaches.

The recent Atlas system by Watson et al. applies machine learning to the assertion generation problem [13]. 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 can recreate the original assertion. They achieved a 31% accuracy of recreating the original assertion on the first assertion generated, with a nearly 50% of re-creating the assertion in the top five generations. Atlas has the benefit of achieving reasonably high accuracy without the runtime associated with a dynamic analysis. However without runtime information it cannot be guaranteed that generated values are accurate to underlying behaviour.

Song, Thummalapenta, and Xie developed UnitPlus, a tool for recommending test and assertion code for developer selected methods [11]. UnitPlus does a static analysis of the class under test to determine which methods from the class are state-modifying and which are observers. By observing which methods modify or observe the same fields, it can recommend appropriate observer methods for a state-modifying method selected by the developer. UnitPlus can then potentially use a string of state-modifying method calls to set up 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 assertion code. However without dynamic information any non trivial expected value would be needed to be provided by the developer.

Xie combines the state-modifying and observer method analysis with dynamic runtime information for the Orstra test generation tool [14]. With a focus on automatically generated test suite, Orstra executes tests and collects all object and observer method states within the test as it runs. As a result Orstra will insert assertions after object creations and updates using appropriate observer methods. What differentiates AutoAssert from Orstra is the time and method of application. Orstra is used over a whole test suite and adds a large number of applicable

assertions after most lines present in the test, without developer involvement. AutoAssert however is designed to be invoked by developers as they work on a test, augmenting the process they would normally follow.

Eclat by Pacheco and Ernst leverages invariant detection to generate assertions [8]. Using the underlying invariant detection system Daikon [4], Eclat observes runtime pre-conditions 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. The tool ZoomIn by Pastore and Mariani also leverages Daikon to identify assertions suited to locating faults, and noting assertions that may be faulty [10]. Additionally invariants may inform future developer written tests. For example if an invariant appears to be unexpectedly restrictive, this may be indicative of an insufficient variety of test inputs. Unlike AutoAssert, assertions generated by Eclat will be generalized to all executions of the method under test, and not tailored to the specific invocation in the test.

Fraser and Arcuri produced EvoSuite which does complete test suite generation [5]. Test body code is created through a mix of methods including symbolic execution and an evolutionary approach. With full access to the tests and all the return values, the problem for assertion generation becomes cutting down the space of possible assertions. EvoSuite uses mutation testing to select the assertions that are most likely to reveal faults. Generating assertions this way provides high quality assertions at the cost of the long runtime associated with mutation testing. EvoSuite's comprehensive test generation but high runtime makes it suitable as a possible replacement for developer involved testing. AutoAssert looks to fill a different use category by assisting developers who are engaging in manual test writing.

Obsidian by Bowring and Hegler is another instance of a developer-in-the-loop system to support developers in test and assertion writing [1]. Obsidian can set up test cases and structure the test cases with helper methods, leaving the developer to specify specific values. Some values can be guessed by statically accessible defaults, such as specific constructor fields, to provide some tests with functional generated assertions. This provides complementary functionality to AutoAssert, providing static test setup but requiring developers to inspect or run the test to determine correct assertion values.

## Chapter 7

# Conclusion

Test case assertions play a fundamental role in automated testing as they check that code under test is behaving as expected. While assertions are numerous, they are also somewhat repetitive and have well-understood structures making them amenable to automatic generation. In this thesis, I investigate how assertions are used in practice through a large empirical study and introduce AutoAssert, a tool to help developers generate assertions for their test cases. AutoAssert is a human-in-the-loop system: developers can explicitly invoke the tool to generate assertions for a variable they want validated in a test case; they can then use their expertise to ensure both that the generated assertions are what they want, and that the asserted values match their expectations. In an empirical study of over 1,000 assertions, AutoAssert was able to generate assertions similar in purpose to those written by developers for the same variable 84% of the time. Through a user study I found that developers found the generated assertions broadly useful and many participants wished to continue to use the tool for their own projects.

# Bibliography

- [1] J. Bowring and H. Hegler. Obsidian: Pattern-Based Unit Test Implementations. *Journal of Software Engineering and Applications*, 2014, Feb. 2014. ISSN 1945-3124. doi:10.4236/jsea.2014.72011. → pages 2, 40
- [2] J. M. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, March 1990. → page 10
- [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)*, pages 500—511, 2019. doi:10.1109/ICSE.2019.00062. → page 1
- [4] 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*, 69(1-3):35–45, Dec. 2007. ISSN 01676423. doi:10.1016/j.scico.2007.01.015. → page 40
- [5] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, page 416, 2011. doi:10.1145/2025113.2025179. → pages 2, 40
- [6] 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 Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 821—830, 2017. doi:10.1145/3106237.3106288. → page 37
- [7] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM (CACM)*, 41(5):81–86, May 1998. doi:10.1145/274946.274960. → page 1

- [8] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *Object-Oriented Programming (ECOOP)*, volume 3586, pages 504–527. 2005. doi:10.1007/11531142\_22. → pages 2, 40
- [9] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, pages 815–816, Oct. 2007. ISBN 978-1-59593-865-7. doi:10.1145/1297846.1297902. → page 2
- [10] F. Pastore and L. Mariani. ZoomIn: Discovering Failures by Detecting Wrong Assertions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 66–76, Florence, Italy, May 2015. IEEE. ISBN 978-1-4799-1934-5. doi:10.1109/ICSE.2015.29. → page 40
- [11] Y. Song, S. Thummalapenta, and T. Xie. UnitPlus: Assisting developer testing in Eclipse. In *Proc. ETX*, pages 26–30, Jan. 2007. doi:10.1145/1328279.1328285. → pages 2, 39
- [12] 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)*, pages 105–115, 2019. doi:10.1109/ICSE.2019.00028. → page 1
- [13] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk. On Learning Meaningful Assert Statements for Unit Test Cases. *arXiv:2002.05800 [cs]*, Feb. 2020. doi:10.1145/3377811.3380429. → pages 2, 37, 39
- [14] T. Xie. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In D. Thomas, editor, *Object-Oriented Programming (ECOOP)*, pages 380–403, 2006. doi:10.1007/11785477\_23. → pages 2, 38, 39
- [15] T. Xie, N. Tillmann, and P. Lakshman. Advances in unit testing: Theory and practice. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 904–905, 2016. → page 1
- [16] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 214–224, Bergamo, Italy, 2015. ACM Press. ISBN 978-1-4503-3675-8. doi:10.1145/2786805.2786858. → page 1

## **Appendix A**

# **Consent Forms and Surveys**

This appendix includes the consent form given to participants prior to their participation in the study, and both the pre-task and post-task surveys taken.



---

## Consent Form

### Automatic Assertion Generation

This study examines how developers write tests, and proposes a new tool to assist with generating assertions for unit tests.

This study consists of four parts:

- This consent form
- A short two page pre-survey
- A web-based tool demo
- A short post-survey

Below is the consent form for the study. Following the consent form will be a short survey about demographics and test writing practices, followed by an interactive demo of the AutoAssert tool, then a final followup survey regarding the tool. It should take 20-30 minutes and you can be entered to win one of four \$50 Amazon gift cards.

### Consent Form

Welcome to our study on assertion generation for software tests.

This page provides some details and background information and asks for your consent to participate.

**Principal Investigator:** Dr. Reid Holmes, Dept. of Computer Science, University of British Columbia

(██████████)

**Co-Investigators:** Lucas Zamprogno, graduate student, Dept. of Computer Science, University of British Columbia (██████████)

#### Study Purpose:

This project investigates details about what software developers include tests they write, the process by which they create them, and how a new tool could assist in this process.

Taking part in this study is entirely up to you. You have the right to refuse to participate in this study. If you decide to take part, you may choose to pull out of the study at any time without giving a reason and without any negative impact on you.

#### What you will be asked to do:

For the study, the participants will respond to a questionnaire regarding their process when writing software tests. They will then use an online prototype of an assertion generation tool, before responding to a final questionnaire about their experience using and opinion of the tool. The study should take 20-30 minutes to complete.

#### Known Risks:

There is some risk participants could be identified by survey responses such as years of experience and

known languages, but that risk very minimal. All directly identifying information will be anonymized.

**Potential benefits:**

Participants could learn from reflecting on their own development practices.

**Payment:**

Participants will be entered into a random draw for one of four \$50 CAD Amazon gift cards with a 1/5 chance of getting the draw. Entry into the draw is conditional on successful task completion as a skill-testing question. Participants from Quebec are not eligible for the draw.

**Data, Storage & Confidentiality:**

All data from the questionnaire and tool will be saved in digital form on UBC servers and the co-investigator's laptop. The devices and servers on which data is stored will be password protected and encrypted.

All data stored will be anonymized and will not contain any identifying information.

You will be identified by number or pseudonyms in any internal or academic research publication or presentation. If we choose to use some of your comments, they will be attributed to a participant number or pseudonym.

The raw data can only be used and seen by researchers directly involved in this project.

The data will be stored for five years, after which local copies will be permanently deleted. Copies of the data may remain on archive files kept by the Department of Computer Science, which adhere to the BC Freedom of Information and Protection of Privacy Act. The principal investigator will be responsible for the data during its retainment time. If you revoke consent and upon your request, all your collected data will be removed. Otherwise, we will maintain all data obtained until the moment you revoked consent.

**Use of the Data:**

The results of this study will potentially appear in both internal and external academic research presentations and publications, such as academic journals and conference proceedings.

**Open data access:**

A study pre-print will be published to <https://arxiv.org/>

**Contact for information about the study:**

If you have any questions or desire further information with respect to the study, you may contact Dr. Reid Holmes ( [REDACTED] )

**Who can you contact if you have complaints or concerns about the study?**

If you have any concerns or complaints about your rights as a research participant and/or your experiences while participating in this study, contact the Research Participant Complaint Line in the UBC Office of Research Ethics at [REDACTED] or in long-distance email [REDACTED] or call toll free



---

Taking part in this study is entirely voluntary. You have the right to refuse to participate in this study. If you decide to take part, you may choose to pull out of the study at any time without giving a reason and without any negative impact on you.

By clicking the statement below, you confirm the following statement

☐ I consent to participate in the study and understand I can leave at any time

Ethics ID number H20-02151

---

Powered by Qualtrics

**Figure A.1:** Consent form.

---

### Demographics

What is the gender you best identify with?

- ☐ Man
- ☐ Woman
- ☐ Non-binary
- ☐ Other

How many years of experience writing code do you have?

How many years of experience writing tests do you have?

Do you consider yourself a software professional?

- ☐ Yes
- ☐ No

How familiar are you with JavaScript?

- Not familiar at all      Slightly familiar      Moderately familiar      Very familiar      Extremely familiar
- ☐                      ☐                      ☐                      ☐                      ☐

### Process

This section will ask about your process when writing software tests.

When writing tests, do you typically place assertions:

- ☐ Above code under test (i.e. to check preconditions)
- ☐ Below the code under test (i.e. to check postconditions)
- ☐ Both
- ☐ I don't include assertions in my tests


Please drag and order options from the left to the area on the right to describe the steps you follow when writing tests alongside source code. You do not need to use all options.

Items	Steps taken
Writing the source code	
Iterating on the source code	
Writing the test setup	
Iterating on the test setup	
Writing the test assertions	
Iterating on the test assertions	
Running the test	
Re-running the test	

If the steps above were insufficient to express your process, please provide additional details here.

What process do you use when writing assertions? For example, do you know the assertions you will write before you create a test or do you run the code under test and inspect the output before writing assertions?

Do you use any external tools for test or assertion generation? If yes, please include what tool or approach. If not, please include any reasons you may have for not.



**Footer**

Thank you for completing the pre-survey. Next we will take you to our web-based demo of our prototype so you can try it out. Afterwards you will have an opportunity to provide feedback.

---

Powered by Qualtrics

**Figure A.2:** Pre-survey.



## Software Practices Lab

### Default Question Block

Would you use a tool like AutoAssert if it were available to you in a convenient format (i.e. integrated with an IDE)?

Very unlikely ☐      Somewhat unlikely ☐      Neither likely nor unlikely ☐      Somewhat likely ☐      Very likely ☐

How did you find AutoAssert's runtime performance?

Very poor ☐      Poor ☐      Average ☐      Good ☐      Very good ☐

In terms of assertion quality, how were the assertions generated compared to what you would write manually?


Much worse ☐      Somewhat worse ☐      About the same ☐      Somewhat better ☐      Much better ☐

How were the automatically generated assertions better or worse from what you would normally write?

Can you foresee scenarios in your experience where an assertion generation approach may fail to identify appropriate assertions?



Do you have any other suggestions you think might improve automatically-written assertions, or the notion of automatic assertion generation in general?



Please provide your email address for entry into the gift card draw, and optional opt-in communications below.



Optional opt-ins for future communications

- ☐ I would be interested in participating in future studies about software engineering
- ☐ I would like to receive followup about the results of this study

---

Powered by Qualtrics

**Figure A.3:** Post-survey.



## **Appendix B**

### **User Study Tasks**

This appendix includes all the tasks users would interact with during the user study, including the tutorial task and poison-pill tasks. The before screenshot is the task as it first appears to the user, after shows assertions generated by running the tool against the target variable. The screenshots after generate also serve as additional examples of the assertions generated by AutoAssert.

```

import {expect} from "chai";
import {Main} from "../src/Main";

/*
This section provides instructions and context for the tasks and tool.
The tasks are taken from real projects using the chai assertion library.

Context for the tests will be provided as comments.
Source code can optionally be shown by clicking the 'Toggle Source' button, but should not be needed.
You do not need to modify the test file beyond any changes you would like to make to the generated assertions;
if you approve of any assertion, please leave it in the code.

To use the tool, right click on the variable (the 'target variable') that needs assertions generated for it.
This test will be run, the value inspected, and assertions will be generated and added to the file.
Once the test has been run and assertions have been generated,
you can add/remove/edit assertions until they feel appropriate.

Once you're done you can click Next Task. After the final task you will be taken to the followup survey.
*/

// Describe indicates a test grouping
describe("Tutorial", function () {

  // it() is a single test
  it("Simple test for demonstration", function () {
    // Right click on the target variable 'res' below and select Generate Assertions
    const res = Main.returnArray();
    // Assertions will appear above this line.
  });
});

```

**Figure B.1:** The tutorial task in the user study, before assertion generation.  
This test was created by me for the purpose of this study.

```

describe("Tutorial", function () {

  // it() is a single test
  it("Simple test for demonstration", function () {
    // Right click on the target variable 'res' below and select Generate Assertions
    const res = Main.returnArray();
    expect(res).to.exist; // Exist means neither null nor undefined
    expect(res).to.be.a("array"); // Typecheck similar to typeof operator
    expect(res).to.deep.equal(["foo",null,1]); // "deep" keyword for value equality
    // Assertions will appear above this line.
  });
});

```

**Figure B.2:** The tutorial task in the user study, after assertion generation.

```

'use strict'

const { expect } = require('chai')
const sinon = require('sinon')
const url = require('url')
const nock = require('..')
const got = require('./got_client')
const assertRejects = require('assert-rejects')

require('./setup')

describe('`query()`', () => {
  describe('when called with an object', () => {

    // A mocked endpoint should be able to respond to a simple request
    it('matches a query string of the same name=value', async () => {
      const scope = nock('http://example.test')
        .get('/')
        .query({ foo: 'bar' })
        .reply()

      // Target variable is statusCode
      const { statusCode } = await got('http://example.test/?foo=bar')

      scope.done()
    })
  })
})

```

**Figure B.3:** Task 1 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock.

```

'use strict'

const { expect } = require('chai')
const sinon = require('sinon')
const url = require('url')
const nock = require('..')
const got = require('./got_client')
const assertRejects = require('assert-rejects')

require('./setup')

describe('`query()`', () => {
  describe('when called with an object', () => {

    // A mocked endpoint should be able to respond to a simple request
    it('matches a query string of the same name=value', async () => {
      const scope = nock('http://example.test')
        .get('/')
        .query({ foo: 'bar' })
        .reply()

      // Target variable is statusCode
      const { statusCode } = await got('http://example.test/?foo=bar')
      expect(statusCode).to.exist; // Exist means neither null nor undefined
      expect(statusCode).to.be.a("number"); // Typecheck similar to typeof operator
      expect(statusCode).to.equal(200); // Value equality check

      scope.done()
    })
  })
})

```

**Figure B.4:** Task 1 of 5 primary non-poison-pill tasks in the user study, after assertion generation.

```

'use strict'

const { expect } = require('chai')
const http = require('http')
const sinon = require('sinon')
const nock = require('..')

const got = require('./got_client')
const { startHttpServer } = require('./servers')

require('./setup')

describe('allowUnmocked option', () => {

  // When an endpoint mock is set up, requests to other endpoints should still go through and get proper responses
  it('allow unmocked post with json data', async () => {
    const { origin } = await startHttpServer((request, response) => {
      response.writeHead(200)
      response.write('{"message": "server response"}')
      response.end()
    })

    nock(origin, { allowUnmocked: true })
      .get('/not/accessed')
      .reply(200, '{"message": "mocked response"}')

    // Target variables is body
    const { body } = await got.post(origin, { json: { some: 'data' }, responseType: 'json' })
  })
})

```

**Figure B.5:** Task 2 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock.

```

'use strict'

const { expect } = require('chai')
const http = require('http')
const sinon = require('sinon')
const nock = require('..')

const got = require('./got_client')
const { startHttpServer } = require('./servers')

require('./setup')

describe('allowUnmocked option', () => {

  // When an endpoint mock is set up, requests to other endpoints should still go through and get proper responses
  it('allow unmocked post with json data', async () => {
    const { origin } = await startHttpServer((request, response) => {
      response.writeHead(200)
      response.write('{"message":"server response"}')
      response.end()
    })

    nock(origin, { allowUnmocked: true })
      .get('/not/accessed')
      .reply(200, '{"message":"mocked response"}')

    // Target variables is body
    const { body } = await got.post(origin, {json: { some: 'data' }, responseType: 'json'})
    expect(body).to.exist; // Exist means neither null nor undefined
    expect(body).to.be.a("object"); // Typecheck similar to typeof operator
    expect(body).to.deep.equal({"message":"server response"}); // "deep" keyword for value equality
  })
})

```

**Figure B.6:** Task 2 of 5 primary non-poison-pill tasks in the user study, after assertion generation.

```

'use strict'

const { expect } = require('chai')
const nock = require('..')
const got = require('./got_client')

require('./setup')

describe('defaultReplyHeaders()', () => {

  // Raw headers are an array of strings in [header, value, header, value, ...] order.
  it('when no headers are specified on the request, default reply headers work', async () => {
    nock('http://example.test')
      .defaultReplyHeaders({
        'X-Powered-By': 'Meeee',
        'X-Another-Header': ['foo', 'bar'],
      })
      .get('/')
      .reply(200, '')

    // Target variable is rawHeaders
    const { rawHeaders } = await got('http://example.test/')
  })
})

```

**Figure B.7:** Task 3 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Nock.

```

'use strict'

const { expect } = require('chai')
const nock = require('.')
const got = require('./got_client')

require('./setup')

describe('defaultReplyHeaders()', () => {

  // Raw headers are an array of strings in [header, value, header, value, ...] order.
  it('when no headers are specified on the request, default reply headers work', async () => {
    nock('http://example.test')
      .defaultReplyHeaders({
        'X-Powered-By': 'Meeee',
        'X-Another-Header': ['foo', 'bar'],
      })
      .get('/')
      .reply(200, '')

    // Target variable is rawHeaders
    const { rawHeaders } = await got('http://example.test/')
    expect(rawHeaders).to.exist; // Exist means neither null nor undefined
    expect(rawHeaders).to.be.a("array"); // Typecheck similar to typeof operator
    expect(rawHeaders).to.deep.equal(["X-Powered-By", "Meeee", "X-Another-Header", ["foo", "bar"]]);
  })
})

```

**Figure B.8:** Task 3 of 5 primary non-poison-pill tasks in the user study, after assertion generation.

```

"use strict";

const typeset = require("../src/index");
const expect = require("chai").expect;

function quotes(html) {
  return typeset(html, {
    enable: ["quotes"],
  });
}

describe("Quotes", () => {

  it("should replace quotes by unicode beginning and ending quotes characters", () => {
    const html = '<p>"Hello," said the fox.</p>';
    // Target variable is 'updated'
    const updated = quotes(html);
  });
});

```

**Figure B.9:** Task 4 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Typeset.

```

"use strict";

const typeset = require("../src/index");
const expect = require("chai").expect;

function quotes(html) {
  return typeset(html, {
    enable: ["quotes"],
  });
}

describe("Quotes", () => {

  it("should replace quotes by unicode beginning and ending quotes characters", () => {
    const html = '<p>"Hello," said the fox.</p>';
    // Target variable is 'updated'
    const updated = quotes(html);
    expect(updated).to.exist; // Exist means neither null nor undefined
    expect(updated).to.be.a("string"); // Typecheck similar to typeof operator
    expect(updated).to.equal("<p>“Hello,” said the fox.</p>"); // Value equality check
  });
});

```

**Figure B.10:** Task 4 of 5 primary non-poison-pill tasks in the user study, after assertion generation.



```

"use strict";

const typeset = require("../src/index");
const expect = require("chai").expect;

function lig(html) {
  return typeset(html, {
    enable: ["ligatures"],
  });
}

describe("Ligatures", () => {

  it("should replace (fi) by (fi)", () => {
    const html =
      "<p>A file folder</p>";
    // Target variable is 'withLigature'
    const withLigature = lig(html);
  });
});

```

**Figure B.11:** Task 5 of 5 primary non-poison-pill tasks in the user study, before assertion generation. This test is from the open-source project Typeset.

```

"use strict";

const typeset = require("../src/index");
const expect = require("chai").expect;

function lig(html) {
  return typeset(html, {
    enable: ["ligatures"],
  });
}

describe("Ligatures", () => {

  it("should replace (fi) by (ſi)", () => {
    const html =
      "<p>A file folder</p>";
    // Target variable is 'withLigature'
    const withLigature = lig(html);
    expect(withLigature).to.exist; // Exist means neither null nor undefined
    expect(withLigature).to.be.a("string"); // Typecheck similar to typeof operator
    expect(withLigature).to.equal("<p>A file folder</p>"); // Value equality check
  });
});

```

**Figure B.12:** Task 5 of 5 primary non-poison-pill tasks in the user study, after assertion generation.

```

const expect = require("chai").expect;
const DateUtil = require("../src/DateUtil");

describe("DateUtil Tests", function () {

  // This method calculates the number of milliseconds elapsed since the beginning of the day
  it("Should get number of milliseconds since the start of the day", function () {
    // Target variable is 'millis'
    const millis = DateUtil.getMillisSinceDayStart();

  });
});

```

**Figure B.13:** Poison-pill task 1 of 2 in the user study, before assertion generation. This test was created by me for the purpose of this study.

```

const expect = require("chai").expect;
const DateUtil = require("../src/DateUtil");

describe("DateUtil Tests", function () {

  // This method calculates the number of milliseconds elapsed since the beginning of the day
  it("Should get number of milliseconds since the start of the day", function () {
    // Target variable is 'millis'
    const millis = DateUtil.getMillisSinceDayStart();
    expect(millis).to.exist; // Exist means neither null nor undefined
    expect(millis).to.be.a("number"); // Typecheck similar to typeof operator
    expect(millis).to.equal(82927826); // Value equality check

  });

});

```

**Figure B.14:** Poison-pill task 1 of 2 in the user study, after assertion generation. Here users should notice that the equality assertion is not a good fit for a result that should change every millisecond.

```

const expect = require("chai").expect;
const UUIDUtil = require("../src/UUIDUtil");

describe("UUID Tests", function () {

  // Part of the unique identifier should be fixed according to the options
  // The other parts of the unique identifier should remain random
  // UUIDs are random identifiers with the format XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX where X is a hex character from 0 to f
  it("Should supply a v1 UUID with partial options", function () {
    const options = {
      node: [0x01, 0x23, 0x45, 0x67, 0x89, 0xab]
    };
    // Target variable is 'uuid'
    const uuid = UUIDUtil.generate("v1", options);

  });

});

```

**Figure B.15:** Poison-pill task 2 of 2 in the user study, before assertion generation. This test was created by me for the purpose of this study.

```

const expect = require("chai").expect;
const UUIDUtil = require("../src/UUIDUtil");

describe("UUID Tests", function () {

  // Part of the unique identifier should be fixed according to the options
  // The other parts of the unique identifier should remain random
  // UUIDs are random identifiers with the format XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX where X is a hex character from 0 to f
  it("Should supply a v1 UUID with partial options", function () {
    const options = {
      node: [0x01, 0x23, 0x45, 0x67, 0x89, 0xab]
    };
    // Target variable is 'uuid'
    const uuid = UUIDUtil.generate("v1", options);
    expect(uuid).to.exist; // Exist means neither null nor undefined
    expect(uuid).to.be.a("string"); // Typecheck similar to typeof operator
    expect(uuid).to.equal("69abb510-2f72-11eb-9b2e-0123456789ab"); // Value equality check

  });

});

```

**Figure B.16:** Poison-pill task 2 of 2 in the user study, after assertion generation. Here users should notice that the equality assertion is not a good fit for an identifier that should be unique every call via randomness.

## **Appendix C**

# **Survey Results**

This appendix contains all the raw data analyzed in the user study. Note that some questions included on the surveys were not used in the final analysis and were not included in the results presented here. All data addressed in this thesis, either summarized or directly, is present.

### **C.1 Pre-task Survey Responses**

Q8 - What process do you use when writing assertions? For example, do you know t...	Q16 - Do you consider yourself a software professional?	Q4 - How familiar are you with JavaScript?	Q6 - This section will ask about your process when writing software tests. When...	Q1 - What is the gender you best identify with?	Q2 - How many years of experience writing code do you have?	Q3 - How many years of experience writing tests do you have?	Q9 - Do you use any external tools for test or assertion generation? If yes, ple...
I tend to have assertions in mind when writing my source code.	Yes	Slightly familiar	Both	Man	13	4	
If I'm doing TDD, I know what the assertions should say because I'm using them to define the behaviour of the code. If I'm writing tests for existing code that I understand well, I will write the assertions based on my knowledge. If I'm writing tests for existing code that I don't understand well, I will often write a test that asserts something I know isn't true (e.g. output should be abcd) just so I can see what it fails with - that'll show me the real output.	Yes	Moderately familiar	Below the code under test (i.e. to check postconditions)	Man	19	8	I don't - I don't know of any such tools.
It depends on the application under test. If I know the expected output then I can write it before running the test.	Yes	Moderately familiar	Below the code under test (i.e. to check postconditions)	Man	5	10	I use Chai assertion library
A bit of both, but I try to write the assertions before running the code.	Yes	Very familiar	I don't include assertions in my tests	Man	5	2	I don't use any tooling for automating test assertion generation, because I have no experience with a tool which would allow for assertion specification that would be sufficiently more advanced than writing tests myself

I write assertions when writing the tests, some tests are just assertions.	Yes	Extremely familiar	Both	Man	10	6	No, I use testing frameworks and libraries, but not auto generation.
Usually I will know the assertions before creating the test	No	Moderately familiar	Below the code under test (i.e. to check postconditions)	Man	5	4	I have never used external tools to help with testing
I create the tests and assertions before running any sort of code.	Yes	Very familiar	Below the code under test (i.e. to check postconditions)	Man	6	3	Depends on what is being tested, otherwise it would be manually created tests.
I usually do TDD and know some basic assertions beforehand since I've spec'd the API for the object under test. Then I iterate from simple to complex (unit to iteration tests)	Yes	Very familiar	Both	Man	20	10	I use snippets to generate boilerplate but not any actual tests or assertions, reason being that I don't know of any tools, but I'd give some a test drive for sure.
Usually by the time I am writing my tests, I am pretty sure the code is mostly correct. I write tests for a few reasons: 1. to help modularize and simplify the code under test itself. By writing the tests, it forces me to think about whether or not the code itself is good quality. 2. to investigate and verify edge cases and that error conditions are properly handled. 3. to ensure that once the behaviour is written, it is guaranteed to stay the same over time (ie-testing for regressions). 4. to document the code under test and illustrate one way to use it. Usually, I have an idea of the assertions I am writing beforehand. Though if the assertion is on a large object, then sometimes I inspect and verify the output before converting that into an assertion.	Yes	Extremely familiar	Below the code under test (i.e. to check postconditions)	Man	25	12	

I generally have an idea of which assertions/conditions I want to test.	Yes	Slightly familiar	Below the code under test (i.e. to check postconditions)	Woman	15	10	Haven't use any tool
That also depends a lot on the context. Sometimes I will just run the test and step in with the debugger to understand better what I will have to assert. Sometimes I will know immediately what assertions I will need and then this step is not necessary. It depends a lot on the complexity of the code I would say.	Yes	Extremely familiar	Both	Man	13	8	Nope. Actually never thought about that really.
I usually write an assert I know will fail, then inspect the actual output after running the test. I usually try to write a narrow assert which might require thinking about what part of the output is static and what could change (e.g., validating the properties of an object but not the actual values).	No	Very familiar	Below the code under test (i.e. to check postconditions)	Man	10	4	No, I've never thought to look for any such tool. Not sure how good they would be: I can craft assertions that might survive changes to the code I know I will make in the future but automatically generated test might need to be constantly updated (e.g., by asserting on structure rather than a specific instance).
Both. If the intent of the test and code is clear, then writing the assertions as part of writing the test makes sense (API testing, for example). In cases where side-effects, or undocumented behaviour is being tested, I will observe the state of the software under test and write the assertions afterwards.	Yes	Slightly familiar	Below the code under test (i.e. to check postconditions)	Man	25	20	I generally write the tests by hand using frameworks like JUnit, Pytest, or UI automation tools like UIAutomator or Espresso. I don't use a tool that generates tests, other than the basic skeleton.
Both processes mentioned in the above example.	No	Not familiar at all	Both	Man	10	10	No, I am not a professional programmer.

I do write the assertions first as a specification for what the code must do.	Yes	Moderately familiar	I don't include assertions in my tests	Man	15	10	No, they might not be precise enough.
I write the initial assertions to specify what I expect to happen in the scenario. On the other hand, subsequent adjustments to the test are sometimes made based on the output.	No	Slightly familiar	Below the code under test (i.e. to check postconditions)	Man	25	10	No. A lot of the tests are small-scale integration tests, and I'm thinking about various tools which are essentially interpreters - generating appropriate inputs would be difficult for such tools.
I try to write whole test case before running it	No	Moderately familiar	Below the code under test (i.e. to check postconditions)	Man	8	2	I briefly used junit, now i don't do anything in java so i don't. I'm not familiar with any other tools
Depends on how familiar I am with the domain of the problem I'm coding for. If I know beforehand what is the expected outcome or if the outcome is defined by a set of requirements, I write assertions first. Otherwise, I inspect the output before writing my assertions.	Yes	Slightly familiar	Both	Woman	2	1	I mostly use Python and the unittests lib in the PyCharm IDE, and I haven't felt the need to have an external tool yet.
I actually follow both the approaches. Sometimes when I am testing a helper method e.g., for checking zip code is valid or not I already know the output for a given invalid zip code so I write the assertion beforehand. However, when I am writing assertions for complicated methods that returns an object e.g., a person object that contains lots of information like name, address etc. I would run the code inspect the output and then write assertion against it given that the output of the code is the expected result.	Yes	Moderately familiar	Both	Man	3	1.5	No.
Mostly before, depending on the complexity. But if done before, I make sure to double check the values.	Yes	Very familiar	Both	Man	8	4	Never had a need



A mix of both, depends on the complexity of the test. In case I write the assertions after, I make sure to go through the output to double check it is correct.	Yes	Extremely familiar	Both	Man	8	4	Never felt a need
Depends on the complexity of what I am testing. If it is easy to see what the correct result is, then I know about it in advance. If it is something more complex, I run the code and then verify that the result makes sense before writing assertions.	No	Very familiar	Below the code under test (i.e. to check postconditions)	Man	10	6	Would consider this if the project I'm working on was very large. Otherwise I'd feel that the effort for learning how to use a new tool is not worth it compared to just writing things from scratch.
I start with something simple as assert is not None and assert some size and as I have a better idea of the desired output, I add more assertions If it's a functionality that will change the system behaviour and I already know the output, I always start with the full assertions beforehand	Yes	Moderately familiar	Both	Man	8	8	none

## C.2 Post-task Survey Responses

Q4 - How were the automatically generated assertions better or worse from what y...	Q5 - Can you foresee scenarios in your experience where an assertion generation...	Q1 - Would you use a tool like AutoAssert if it were available to you in a conve...	Q2 - How did you find AutoAssert's runtime performance?	Q3 - In terms of assertion quality, how were the assertions generated compared t...	Q6 - Do you have any other suggestions you think might improve automatically-wri...
I typically don't use a test suite, and have asserts in my source code that help me track down issues later on. Asserts through a test system feel better.		Neither likely nor unlikely	Average	Somewhat better	
They were very thorough, which encouraged me to include more of them, although you'll notice that there were some I almost always (but not always!) removed.	Failing to identify negative cases - i.e. not just that the return value should be a certain form, but specify what characteristics it shouldn't have. To be honest, this would probably require additional tests rather than just additional assertions...	Very likely	Very good	Somewhat better	Not sure, sorry
The assertions were generally overly verbose. Doing an existence check on every single call seems superfluous		Somewhat likely	Good	Somewhat worse	
More explicit - a null check may be unnecessary if there's already a deep equal check right after. Too many assertions may cause runtime issues with running tests if there are many, but in a small project it's a good thing to have. Also, comments explaining the assertions and reasoning behind why the expected output is such is good.	The example with the time from ... generated an assertion that expected a static value, where in reality the value would change over time, so a static assertion in that case would fail.	Very likely	Poor	Somewhat better	It would be interesting to see if it's possible to automatically generate test files from code directly. The demo shows that it's possible to create assertions from specified variables, but it should be a similar idea to iterate through a file and generate test cases for each function. This way it would promote testing on functions people may otherwise neglect.
It would generate assertions that I might be too lazy to write myself.	Yes when the variable might be dependant on changing circumstance outside of the code	Somewhat likely	Average	Somewhat better	It would be useful if the user could select the types of assertions to generate so they do not have to remove the same assertions in a block of tests where it is unnecessary.

It's not useful to generate 3 assertions (ie- emptiness check, type check, and equality check) for each variable. The first two are redundant and can be removed. You need to be careful about respecting formatting of the user's editor (eg- single/double quotes, indentation, line wrapping).	Sometimes strict equality is important, sometimes it's important that an object is deep equal, but not strict equal. Will this work with asynchronous code? Assertions inside promises?	Somewhat likely	Average	Somewhat worse	It would be great if you could identify code clones and automatically extract them into variables.
They were about the same. I couldn't think of additional assertions I would have wanted to test.	Would be curious to see how it behaves with null/undefined objects.	Somewhat likely	Good	About the same	
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. Maybe if the assertions were being automatically generated it would be helpful since I maybe wouldn't be as familiar. They might also be good if I was running someone else's tests.	Just instances where there would be randomness.	Somewhat likely	Good	About the same	I would if there would be some way to give feedback about the confidence of the assertions. Like maybe it could try regenerating the same assertion then warn the user if the value changes (I'm thinking of the millisecond task)so they can manually adjust the test. Maybe there could be an adjustable value for strictness or number of assertions: initially only check the type, then once I'm finished my code changes, apply the value assertions too.
Functions whose output is random (like uuid) or time-dependent (like milliseconds) aren't well-suited for this framework.	Yes, these tests only validate happy-paths, so handling error and edge cases are needed.	Very likely	Good	Somewhat worse	This is a great idea, but may offer the developer a false sense of security. If it could be coupled with code coverage, then it would be much more powerful.
It's definitely better than /no assertions/. However it does not take into account the individual invariants (like the UUID example), and may lead users to a false sense of security about their assertions (though it can also be useful for reflection on why the automatically generated assertions are incorrect).	Yes: Effectful computations (like require(/.setup)). Promise trigerring invariants, randomized outputs (as in examples)	Somewhat likely	Good	Somewhat worse	I wanted to re-trigger assertion generation to see different random outputs (UUIDs) and the system failed. It would be nice to be able to retrigger on purpose.

It's much more convenient for remembering corner cases (e.g. testing for undefined), but typically too specific on equality constraints. For array-like data I suspect abstracting these constraints to length constraints or even just non-emptiness would be more useful in most cases.	Where the functionality of the code called is sufficiently complex, it's unlikely one can say much in terms of deep equality constraints; constraints about not being undefined or others about the expected structure of the data are more likely to fit.	Very likely	Good	About the same	Give options between value constraints of various degrees of abstraction (deep equals vs. some elements are equal vs. lengths are equal vs. length is at least vs. length is non-zero)
I usually only check for value, and don't check type	It seems like function returning milliseconds from start of the day was a good example, but it would be hard for a tool to guess what is a pure function and what isn't	Somewhat likely	Good	Somewhat better	Maybe the whole test cases could be generated?
I believe the generated assertions were about the same as the ones I would normally write.	I haven't encountered this problem before, but the assertion generation approach may fail in a scenario where only a partial match of a string is required (the equivalent of <code>'''assert 'utf-8' in 'html; charset=utf-8''''</code> in python).	Very likely	Good	About the same	
worse: for non-deterministic scenarios like a method that get times in millis better: I gain time on default http stuff like asserting that a request/response status is 200 or something similar	complex compiler scenarios. If I run a program that outputs a compiled program, it's likely that it can generate automatic assertions for the compilation 1 success 0 error but not for the actual file with the program that I want to compare. More complex cases follow a similar logic.	Very likely	Good	Somewhat better	Generating assertions based on source code file. When I am in a certain source file and I click generate assertion for a field or method in that file, it would produce something in the current test that I am at. I originally thought that this was how the tool was supposed to work

It is about the same with the exception of a few assertions that I would have added.	When the return type is any the assertion generation approach might fail	Very likely	Good	About the same	Checking the length of the expected content
They were similar, but I have rarely written tests as simple as this so it's hard to judge. Some things (like always check type) could be skipped.	I wonder how it'd perform on more complicate situations. All the examples given followed the exact same pattern.	Neither likely nor unlikely	Very good	About the same	
They do cover cases that I might miss (e.g. the exists checks), but they also require me to think about them since not all of them make sense (e.g. the date one)	It seems like the assertions only work if the checked value is deterministic and constant. If there is anything even slightly more complex to check they won't work anymore.	Somewhat likely	Very good	About the same	I feel like running the code multiple times and trying to factor out what the common components of all the results are would be a good start. I feel like I'd be more comfortable if the generated assertions are at least always correct (then I could use them as a starting point for adding my own ones, similar to generating constructors/getters/.. in a Java IDE.

## Appendix D

# Assertion Generation Details

This appendix includes examples of assertions generated, and not generated, by AutoAssert as well as the code injected to perform the logging step of generation.

```
expect(result).to.be.null;  
expect(result).to.be.undefined;  
expect(result).to.exist;  
expect(result).to.throw;  
expect(result).to.not.throw;  
expect(result).to.equal(value);  
expect(result).to.deep.equal(value);  
expect(result).to.have.length(value);  
expect(result).to.be.a(value);  
expect(result).to.be.true;  
expect(result).to.be.false;
```

**Listing D.1:** All assertion forms potentially generated by AutoAssert

```

[{ // Result 1
  type: 'number',
  value: 42
},
{ // Result 2
  type: 'number',
  value: 42
}]

// Generated assertions
expect(result).to.exist;
expect(result).to.be.a("number");
expect(results).to.equal(42);

```

**Listing D.2:** A typical double run where the value is the same on both runs.  
All appropriate assertions will be generated.

```

[{ // Result 1
  type: 'number',
  value: 42
},
{ // Result 2
  type: 'number',
  value: -1
}]

// Generated assertions
expect(result).to.exist;
expect(result).to.be.a("number");

```

**Listing D.3:** A double run where the value is different. In this case the assertions for existence and type are kept from the typical scenario, but the exact equality check is removed.

```

[{ // Result 1
  type: 'number',
  value: 42
},
{ // Result 2
  type: 'string',
  value: "foo"
}]

// No assertions generated

```

**Listing D.4:** A double run resulting in completely different types. In this instance an exist check could still apply, but I decided that if even the type is variable it is safer to assume it could be anything.

Numeric  
Value: 4  
Assertion: `expect(index).to.be.above(-1);`  
Problem: Cannot determine appropriate range and comparison values from a single value.

Inclusion (Array)  
Value: `["Canada", "Australia", "New Zealand", "United Kingdom"]`  
Assertion: `expect(countries).to.include("Canada");`  
Problem: Cannot infer which element is the one of importance.

Inclusion (String)  
Value: `"www.ubc.ca"`  
Assertion: `expect(url).to.include(".");`  
Problem: Cannot infer which substring is the one of importance.

Property  
Value: A large object of file metadata (too big to list)  
Assertion: `expect(file).to.have.property("lastModified");`  
Problem: Cannot infer which property is the one of importance.

Truthiness  
Value: `"hunter2"`  
Assertion: `expect(password).to.be.ok;`  
Problem: Truthiness is a messy metric compared to equality and `boolean` checks. Chai itself recommends against its use.

Calls  
Value: `function addOne(x) {return x+1}`  
Assertion: `expect(func).to.have.been.called.with(1);`  
Problem: Requires identifying the value as a `function` before runtime, and instrumenting that as well.

Patterns  
Value: `"Hello world"`  
Assertion: `expect(greeting).to.match(/^Hello/);`  
Problem: Cannot determine appropriate pattern.

**Listing D.5:** The categories not support by AutoAssert, examples values and associated assertion, and the problems with each category that lead to it not being supported.



```
function elemUnderTestGenerator(elem) {
  const res = {};
  const type = typeof elem;
  res["type"] = type;
  switch (type) {
    case "boolean":
    case "number":
    case "string":
      res["value"] = elem;
      break;
    case "symbol":
      res["value"] = elem.toString();
      break;
    case "undefined":
      res["value"] = null;
      break;
    case "function":
      let numArgs = elem.length;
      res["args"] = numArgs;
      if (numArgs === 0) {
        let throws = false;
        try {
          elem();
        } catch (e) {
          throws = true;
        }
        res["throws"] = throws;
      }
      res["value"] = elem.toString();
      break;
  }
}
```

**Figure D.1:** JavaScript code injected into projects to log the resulting value. Some types such as promises did not have unique implementations on the generation side at the time of writing. (1/2)

```

        case "object":
            if (elem === null) {
                res["type"] = "null";
                res["value"] = null;
            } else if (Array.isArray(elem)) {
                res["type"] = "array";
                res["value"] = elem;
                res["length"] = elem.length;
            } else if (elem instanceof Set) {
                res["type"] = "set";
                res["value"] = Array.from(elem);
                res["length"] = elem.size;
            } else if (typeof elem.then === 'function') {
                res["type"] = "promise";
            } else if (elem instanceof Error) {
                res["type"] = "error";
                res["value"] = elem.message;
            } else {
                res["value"] = elem;
                res["methods"] = [];
                for (const key of Object.keys(elem)) {
                    if (typeof elem[key] === "function") {
                        res["methods"].push(key);
                    }
                }
            }
        }
    }
    return res;
}

require("fs").writeFileSync(
    __dirname + IsolatedAssertionGeneration.logFile,
    JSON.stringify(elemUnderTestGenerator(varName)));

```

**Figure D.2:** JavaScript code injected into projects to log the resulting value. Some types such as promises did not have unique implementations on the generation side at the time of writing. (2/2)