PathSpotter: Exploring Tested Paths to Discover Missing Tests

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

ABSTRACT

When creating test cases, ideally, developers should test both the expected and unexpected behaviors of the program to catch more bugs and avoid regressions. However, the literature has provided evidence that developers are more likely to test expected behaviors than unexpected ones. In this paper, we propose PathSpotter, a tool to automatically identify tested paths and support the detection of missing tests. Based on PathSpotter, we provide an approach to guide us in detecting missing tests. To evaluate it, we submitted pull requests with test improvements to open-source projects. As a result, 6 out of 8 pull requests were accepted and merged in relevant systems, including CPython, Pylint, and Jupyter Client. These pull requests created/updated 32 tests and added 80 novel assertions covering untested cases. This indicates that our test improvement solution is well received by open-source projects.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Runtime environments**.

KEYWORDS

software testing, test quality, runtime monitoring, python

ACM Reference Format:
Andre Hora. 2024. PathSpotter: Exploring Tested Paths to Discover Missing Tests. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663816

1 INTRODUCTION

Having a good test suite is fundamental to ensuring software quality and sustainable software evolution [4, 10, 28]. When creating test cases, developers should focus on testing both the expected and unexpected behaviors of the program [2, 18, 24]. The expected behavior represents the normal execution, in which nothing goes wrong [18], while the unexpected behavior refers to the abnormal execution. Test suites should ideally test both scenarios to catch more bugs and protect against regressions [1, 15, 16, 28].

In practice, it is well-known that developers are more likely to test expected behaviors than unexpected ones [2, 3, 5, 6, 8, 12, 17, 19, 25, 26]. This happens because the expected behavior of the program is often simpler to test. Another factor is that developers may lack test expertise, focusing on only testing the "happy cases" [8].

In this paper, we propose PathSpotter, a tool to automatically identify tested paths and support the detection of missing tests. A tested path represents a set of input values that will make the method behave in the same way.[1] Based on PathSpotter, we provide an approach to guide us in detecting missing tests. Particularly, we propose to explore the most and least tested paths of a method to reveal untested cases. To evaluate it, we improved the test suites of real-world systems and submitted the changes via pull requests. As a result, the approach successively guided us in improving the test suites of relevant systems, including CPython, Pylint, and Jupyter Client. We have 6 out of 8 pull requests accepted, created/updated 32 test methods, and added 80 novel assertions covering untested cases. PathSpotter is publicly available at: https://github.com/andrehora/pathspotter. Screencast: https://youtu.be/SeHewe34Q24.

Contributions. The contributions of this paper are twofold. First, we provide PathSpotter, a tool to automatically identify tested paths and support the detection of missing tests. (Section 3). Second, we propose and evaluate an approach to improve test suites (Section 4).

2 MOTIVATION

A good test suite should work as a protection against regressions [1, 15, 16, 28]. Unfortunately, not all test suites work as real protection against regressions because they may focus on expected behaviors than unexpected ones [2, 3, 5, 6, 8, 12, 17, 19, 25, 26]. In this case, ideally, developers should strive to improve their tests. Next, we describe a real-world scenario of test improvement.

Consider the method `cell_len` (see Figure 1) of the popular project Rich. This public method provides the number of cells required to display a text and has two behaviors: one for short texts (*i.e.,* < 512 characters, which uses cache) and the other for long texts (*i.e.,* ≥ 512 characters, which does not use cache).

```
29  def cell_len(text: str, _cell_len: Callable[[str], int] = cached_cell_len) -> int:
30      """Get the number of cells required to display text.
31
32      Args:
33          text (str): Text to display.
34
35      Returns:
36          int: Get the number of cells required to display text.
37      """
38      if len(text) < 512:
39          return _cell_len(text)
40      _get_size = get_character_cell_size
41      total_size = sum(_get_size(character) for character in text)
42      return total_size
```

Figure 1: Method `cell_len` of project Rich.

[1] Example: https://andrehora.github.io/tested_paths_dataset/report_html/calendar/calendar.monthrange.html

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663816

As `cell_len` is a public API, ideally, both behaviors should be tested. However, originally, only the long text behavior was directly tested by Rich's test suite, that is, there exists a test called `test_cell_len_long_string` dedicated to cover the long text behavior. In contrast, there is no test focused on the short text behavior (its lines of code are only indirectly covered by other tests, thus, the line and branch coverage of method `cell_len` is 100%). Notice that one important best practice in software testing recommends that we focus on testing behaviors [28]. Thus, there is a missing test dedicated to the short text behavior. In this case, we are left with one question: how can we improve the tests of `cell_len`? Unfortunately, we cannot rely on coverage metrics because method `cell_len` has already 100% of line and branch coverage.

To overcome this problem, we propose an approach to improve test suites based on the exploration of the tested paths. With the support of PathSpotter, we explore the most and least tested paths of a method to reveal possibly untested cases. Based on the proposed approach, we improved the tests of Rich's `cell_len` and submitted the changes via pull request.

Our test improvement was accepted and merged in Rich (PR 2786), as presented in Figure 2. The original test covering the long text behavior is presented in lines 4-6. We updated the existing test to cover a boundary case (lines 7-8) and we created a new test to cover the untested behavior (lines 11-15). In Section 4, we provide more details about the proposed approach to improve tests.

```
4    def test_cell_len_long_string():
5        # Long strings don't use cached cell length implementation
6        assert cells.cell_len("abc" * 200) == 3 * 200
7  +     # Boundary case
8  +     assert cells.cell_len("a" * 512) == 512
9  +
10 +
11 + def test_cell_len_short_string():
12 +     # Short strings use cached cell length implementation
13 +     assert cells.cell_len("abc" * 100) == 3 * 100
14 +     # Boundary case
15 +     assert cells.cell_len("a" * 511) == 511
```

Figure 2: PR 2786 accepted in Rich with the improved tests of `cell_len`. Original test: lines 4-6. Improved tests: lines 7-15.

3 PATHSPOTTER

Figure 3 presents an overview of the technique that supports PathSpotter. First, we execute an instrumented version of the tests, collecting call states from the execution traces (Section 3.1). Next, based on the collected call states, we detect the tested paths (Section 3.2) and compute the ranking of tested paths (Section 3.3). Finally, Section 3.4 presents implementation notes.

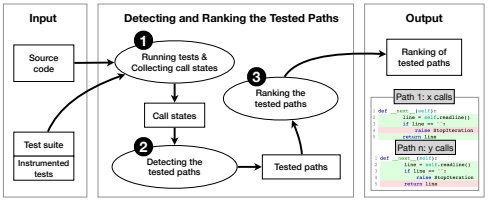

Figure 3: Overview of the technique to identify tested paths.

3.1 Running Tests and Collecting Call States

First, we need to run the test suite and collect relevant information about the system execution. For this purpose, we execute an instrumented version of the test suite that monitors the tests and collect data from the execution trace.

For each call of method m, we collect its call state. A call state cs_m of method m is formed by: (1) the executed lines of code, (2) the parameter values, (3) the return value, and (4) the thrown exception if some exception happens during the call. We collect the executed lines of code because they are the basis to compute tested paths. We collect the inputs, outputs, and thrown exceptions because they allow us to better understand the program's behavior. After running all tests, for each executed method m, we have a set of call states.

3.2 Detecting the Tested Paths

A *tested path* of an executed method m is defined as a sorted set of executed lines of code of m. For each executed method m, we compute its tested paths based on its set of call states. A tested path tp_m of method m is formed by a triple: (1) unique lines of code, (2) path frequency, and (3) path ratio. The *unique lines of code* is computed as follows: for each call state cs_m of m, we collect the executed lines of code as a sorted set. We use a sorted set to avoid line duplication due to the execution of loops, thus, lines that are executed multiple times (due to loops) are counted once. The *path frequency* and *path ratio* are simply the absolute and relative frequencies of the tested path. As a result, we have a set of methods, each with at least one tested path.

3.3 Ranking the Tested Paths

Lastly, for each method m with one or more tested paths tp_m, we sort their paths in descending order of path frequency, creating a *ranking of tested paths*. Thus, the most executed paths are top-ranked, while the least executed ones are bottom-ranked.

3.4 Usage and Implementation Notes

3.4.1 Usage. PathSpotter implements the proposed technique to detect and rank tested paths. PathSpotter also provides metrics and code visualization to support understanding the tested paths of a method, including the most and least tested paths as well as their executed lines of code, inputs, outputs, and thrown exceptions. It can be used by researchers and practitioners to comprehend, maintain, create, and improve test cases.

Figure 4 presents an overview of PathSpotter. For a given method, it shows (1) the method overview and (2) the path details. The *path details* present the tested paths sorted and contain (2.1) the path frequency and ratio, (2.2) the path parameter values and their frequency between parentheses, (2.3) the path return values and their frequency, and (2.4) the executed path code. PathSpotter generates HTML reports for the whole project (for example, calendar,[2] csv,[3] and gzip[4]), which provides reports for individual methods.

3.4.2 Implementation Notes. PathSpotter is implemented in Python and targets Python systems. PathSpotter is implemented on the top

[2] calendar: https://andrehora.github.io/pathspotter/examples/report_html/calendar
[3] csv: https://andrehora.github.io/pathspotter/examples/report_html/csv
[4] gzip: https://andrehora.github.io/pathspotter/examples/report_html/gzip

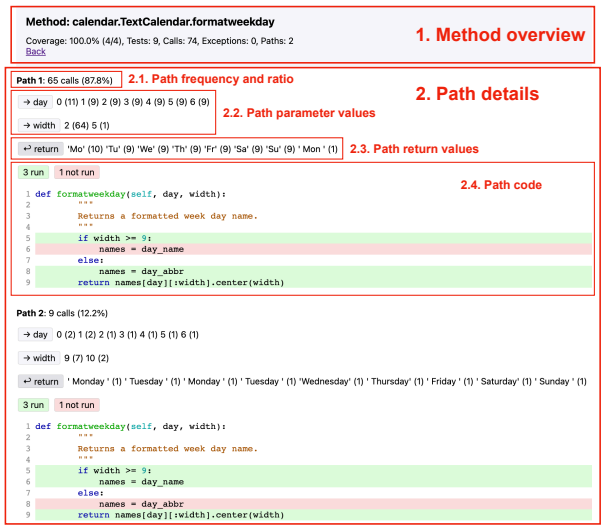

Figure 4: Overview of PathSpotter.

of SpotFlow [14], a tool that supports runtime analysis of Python programs.[5] To instrument the test and collect runtime data, SpotFlow sets the test suite's trace function [21, 23] and monitors events (*e.g.*, calls, line execution, etc.) to collect call states and to detect and rank tested paths. The overhead added by SpotFlow is not irrelevant [14] but is in line with similar runtime tools [9, 11].

4 DISCOVERING MISSING TESTS

Based on PathSpotter, we propose an approach to guide us in improving test suites. tested paths. This approach has the following steps: (1) detect the candidate method, (2) explore and understand the tested paths, (3) identify relevant untested inputs and outputs, (4) explore the tests that exercise the candidate method, and (5) create/update the tests. Next, we detail and evaluate the approach.

4.1 Detect the Candidate Method

The first step is to spot the method that is candidate to be further tested. One solution is to select methods with a low top n path ratio. Those methods are likely to have infrequently tested paths, which can potentially be a first source of exploration.

Example: As an example, we will analyze the method `formatweekday`[6] of the calendar Python Standard Library. This method returns a formatted weekday based on the `day` and `width` arguments and its top n path ratio is only 12%.

4.2 Explore and Understand the Tested Paths

Next, we explore and understand the tested paths of the candidate method. For this purpose, we rely on PathSpotter (see Section 3.4 for more details) to identify and explore the tested paths as well as the parameter and return values that cause the paths to be executed. PathSpotter can be used to help developers identify equivalence classes and their boundaries [1, 15, 20]. An equivalence class is a set of input values that will make the program behave in the same way and it has boundaries with other equivalence classes [15].

Example: Figure 4 presents the tested paths of the method `formatweekday`. This method has 74 calls and two paths: Path 1 with 65 calls (88%) and Path 2 with 9 calls (12%). Next, with the support of PathSpotter, we explore the paths to better understand them.[7]

We can divide the inputs into two equivalence classes: (1) the ones that return the abbreviated weekdays and (2) the ones that return the non-abbreviated weekdays. First, we notice that Path 1 returns the abbreviated weekdays. The `day` parameter values range from 0 to 6, representing the weekday (the value 0 happens 11 times, while the other values happen 9 times). The `width` parameter values are 2 and 5 (the value 2 happens 64 times and the value 5 happens 1 time). The return values are the abbreviated weekdays, like "Mo" and "Tu". The return value "Mo", for example, happens 10 times.

Second, we observe that Path 2 represents a distinct behavior of the method, returning the non-abbreviated weekdays. In this case, the `day` values also range from 0 to 6, but the `width` values are 9 and 10. The return values are the non-abbreviated weekdays, from "Monday" to "Sunday".

4.3 Identify Untested Inputs and Outputs

After exploring and understating the tested paths, we rely on the proposed tool to identify untested inputs and outputs. Specifically, for each tested path, we assess the parameter and return values and frequencies, looking for relevant untested cases, like boundary values [1, 15]. Those untested cases are candidates to be tested.

Example: Consider the method `formatweekday` shown in Figure 4. In Path 1 (the behavior that returns the abbreviated weekdays), the `width` argument is tested only with 2 and 5, missing other values like 1 (the smallest case) and 8 (the boundary case). We can also detect untested values in the outputs. For example, in Path 1, we see that the return values are strings with 2 or 3 characters, thus, we could verify the case the return values have other sizes, like 1.

4.4 Explore the Tests that Exercise the Candidate Method

Next, we identify the tests that exercise the candidate method with the support of the proposed tool. Notice that one or more tests may execute the candidate method, either directly or indirectly. We explore those tests to reason about the need to create new tests (or update existing ones) that cover the untested cases.

Example: Figure 4 shows that nine tests are exercising the method `formatweekday`. We identify that the test `test_locale_calendar_formatweekday` is the one most focused on `formatweekday`, as presented in Figure 5. After exploring this test, we notice only three calls to `formatweekday` (in lines 572, 574, and 576), which test, respectively, the following `width` values: 5, 2, and 10. Recall that, in the previous step, we identified at least two untested inputs that could be tested for the argument `width`: 1 and 8.

Here, we spot another relevant input that is not tested: 9, which is a boundary case for Path 2. Indeed, despite the value 9 is presented in PathSpotter (see the `width` values of Path 2 in Figure 4), this case is *not* covered by the target test.

[5] https://github.com/andrehora/spotflow
[6] https://github.com/python/cpython/blob/4fccf91/Lib/calendar.py#L323-L331
[7] PathSpotter report for the method `formatweekday`: https://andrehora.github.io/pathspotter/examples/report_html/calendar/calendar.TextCalendar.formatweekday.html

Table 1: Summary of the accepted pull requests and the candidate methods.

Accepted PR	System	Example of Created/Updated Tests	Candidate Method	Paths	Calls	Top 1 Path		Top n Path	
						Freq.	Ratio	Freq.	Ratio
3514	BentoML	test_valid_runner_short_tag	validate_tag_str	2	445	441	99.1%	4	0.9%
3521	BentoML	test_rename_fields_remove_only	rename_fields	5	311	289	92.9%	1	0.3%
	BentoML	test_invalid_load_config_file	load_config_file	2	37	36	97.3%	1	2.7%
	BentoML	test_invalid_ip_address	is_valid_ip_address	2	114	112	98.2%	2	1.8%
929	Jupyter Client	test_parse_date_invalid	parse_date	2	102	69	67.6%	33	32.4%
	Jupyter Client	test_extract_dates_from_dict	extract_dates	4	119	91	76.5%	1	0.8%
2786	Rich	test_cell_len_short_string	cell_len	2	47,801	47,800	99.99%	1	0.01%
	Rich	test_pick_bool	pick_bool	2	2,332	2,331	99.99%	1	0.01%
101378	CPython	test_locale_calendar_formatweekday	formatweekday	2	74	65	87.8%	9	12.2%
8159	Pylint	test_unknown_keyword_with_missing_messages	emit_pragma_representer	2	1,349	1,348	99.99%	1	0.01%

```
567     def test_locale_calendar_formatweekday(self):
568         try:
569             # formatweekday uses different day names based on the available width.
570             cal = calendar.LocaleTextCalendar(locale='en_US')
571             # For short widths, a centered, abbreviated name is used.
572             self.assertEqual(cal.formatweekday(0, 5), " Mon ")
573             # For really short widths, even the abbreviated name is truncated.
574             self.assertEqual(cal.formatweekday(0, 2), "Mo")
575             # For long widths, the full day name is used.
576             self.assertEqual(cal.formatweekday(0, 10), "  Monday  ")
577         except locale.Error:
578             raise unittest.SkipTest('cannot set the en_US locale')
```

Figure 5: Test `test_locale_calendar_formatweekday`.

4.5 Create/Update the Tests

Finally, we improve the tests to cover the identified untested values. We can update an existing test method in case there exists a test covering the behavior. We can also create a new test method to cover the untested values in case there is no test covering the behavior.

Example: Figure 6 presents the improved version of the test `test_locale_calendar_formatweekday`. We updated the existing test to cover the identified inputs: 1, 3, 8, and 9. This test improvement was submitted, accepted, and merged in the Python Standard Library (CPython), as presented in PR 101378.

```
567     def test_locale_calendar_formatweekday(self):
568         try:
569             # formatweekday uses different day names based on the available width.
570             cal = calendar.LocaleTextCalendar(locale='en_US')
571             # For really short widths, the abbreviated name is truncated.
572             self.assertEqual(cal.formatweekday(0, 1), "M")
573             self.assertEqual(cal.formatweekday(0, 2), "Mo")
574             # For short widths, a centered, abbreviated name is used.
575             self.assertEqual(cal.formatweekday(0, 3), "Mon")
576             self.assertEqual(cal.formatweekday(0, 5), " Mon ")
577             self.assertEqual(cal.formatweekday(0, 8), "  Mon   ")
578             # For long widths, the full day name is used.
579             self.assertEqual(cal.formatweekday(0, 9), "  Monday ")
580             self.assertEqual(cal.formatweekday(0, 10), "  Monday  ")
581         except locale.Error:
582             raise unittest.SkipTest('cannot set the en_US locale')
```

Figure 6: Improved version of the test `test_locale_calendar_formatweekday` (merged in CPython, PR 101378).

4.6 Evaluation

To evaluate the approach, we propose to improve the test suites of seven relevant and real-world systems: CPython, Rich, Pylint, Jupyter Client, BentoML, Flask, and The Fuck. The test improvement was submitted to those projects via pull requests (PRs) in GitHub and the author had no prior experience with the target projects. We applied the approach described in Sections 4.1 to 4.5 until we submitted 10 PRs (at most 2 pull requests per project to avoid any perception of over-contribution). We recall that having changes accepted to mature systems is not trivial [22]. Well-established projects are more conservative in accepting PRs [7, 13, 27]. Thus, PRs with low-priority contributions are unlikely to be accepted [7].

Finally, we submitted 10 PRs with test improvement and received eight answers. Six out of eight PRs were accepted in BentoML (3514 and 3521), Jupyter Client (929), Rich (2786), CPython (101378), and Pylint (8159). Overall, we created/updated 32 test methods and added 80 novel assertions covering untested cases. The six accepted PRs tested 10 distinct methods, as detailed in Table 1. Due to the space limit, we briefly describe the accepted PRs in Pylint and Rich.

Pylint: PR 8159 created a test for `emit_pragma_representer`. This method parses code pragmas and verifies if they are well-formed. It has 1,349 calls, but the bottom path has only one. This allowed us to identify one missing test for an invalid pragma pattern, leading to the `test_unknown_keyword_with_missing_messages`.

Rich: PR 2786 improved three tests of two methods. In both methods, the bottom paths are rarely executed: 1 out of 47,801 calls in `cell_len` and 1 out of 2,332 calls in `pick_bool`. First, we focused on method `cell_len`, which returns the number of cells required to display a text (see Figure 2). The second method is `pick_bool`; we improved its existing test by covering untested cases, particularly when the list passed as a parameter has exactly one element.

Not accepted pull requests: Two PRs were not accepted. In Pylint (8188), the maintainer explained that the tested code was stable. In Flask (4957), we tested a method via the CLI (command-line interface), but the maintainer suggested testing the method directly.

5 CONCLUSION

In this paper, we presented PathSpotter, a tool to automatically identify tested paths and support the detection of missing tests. The proposed tool has successively guided us in detecting missing tests. Overall, we have 6 out of 8 accepted PRs, created/updated 32 test methods, and added 80 novel assertions covering untested cases. As future work, we plan to evaluate PathSpotter with developers in tasks related to test compression and improvement.

ACKNOWLEDGMENT

This research is supported by CNPq, CAPES, and FAPEMIG.

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Simon and Schuster.

[2] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4925–4946.

[3] Gina R Bai, Justin Smith, and Kathryn T Stolee. 2021. How students unit test: Perceptions, practices, and pitfalls. In *ACM Conference on Innovation and Technology in Computer Science Education*. 248–254.

[4] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[5] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. 2021. How do students test software units?. In *International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 189–198.

[6] Adnan Causevic, Rakesh Shukla, Sasikumar Punnekkat, and Daniel Sundmark. 2013. Effects of negative testing on TDD: An industrial experiment. In *International Conference on Agile Processes in Software Engineering and Extreme Programming*. Springer, 91–105.

[7] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. 2017. A systematic mapping study of software development with GitHub. *IEEE Access* 5 (2017), 7173–7192.

[8] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Conference on Innovation & technology in Computer Science Education*. 171–176.

[9] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 760–771.

[10] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Prentice Hall Professional.

[11] Cormac Flanagan and Stephen N Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 1–8.

[12] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52–81.

[13] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this? evaluating code contributions with language models. In *Working Conference on Mining Software Repositories*. IEEE, 157–167.

[14] Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *International Conference on Software Engineering*. 1–5.

[15] Cem Kaner, Sowmya Padmanabhan, and Douglas Hoffman. 2013. *The Domain Testing Workbook.* Context Driven Press.

[16] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns.* Simon and Schuster.

[17] Laura Marie Leventhal, Barbee M Teasley, Diane S Rohlman, and Keith Instone. 1993. Positive test bias in software testing among professionals: A review. In *International Conference on Human-Computer Interaction*. Springer, 210–218.

[18] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[19] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. 2018. Cognitive biases in software engineering: a systematic mapping study. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1318–1339.

[20] Thomas J. Ostrand and Marc J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Commun. ACM* 31, 6 (1988), 676–686.

[21] Python sys.settrace. November, 2023. https://docs.python.org/3/library/sys.html#sys.settrace.

[22] Mohammad Masudur Rahman and Chanchal K Roy. 2014. An insight into the pull requests of github. In *Working Conference on Mining Software Repositories*. 364–367.

[23] Giles Reger and Klaus Havelund. 2016. What is a trace? A runtime verification perspective. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 339–355.

[24] Stuart C Reid. 1997. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *International Software Metrics Symposium*. IEEE, 64–73.

[25] Iflaah Salman, Burak Turhan, and Sira Vegas. 2019. A controlled experiment on time pressure and confirmation bias in functional software testing. *Empirical Software Engineering* 24, 4 (2019), 1727–1761.

[26] Barbee E Teasley, Laura Marie Leventhal, Clifford R Mynatt, and Diane S Rohlman. 1994. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology* 79, 1 (1994), 142.

[27] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *International Conference on Software Engineering*. 356–366.

[28] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.

Received 2024-01-29; accepted 2024-04-15