SpotFlow: Tracking Method Calls and States at Runtime

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

ABSTRACT

Understanding the runtime behavioral aspects of a software system is fundamental for several software engineering tasks, such as testing and code comprehension. For this purpose, typically, one needs to instrument the system and collect data from its execution. Despite the importance of runtime analysis, few tools have been created and made public to support developers extracting information from software execution. In this paper, we propose SpotFlow, a tool to ease the runtime analysis of Python programs. With SpotFlow, practitioners and researchers can easily extract information about executed methods, run lines, argument values, return values, variable states, and thrown exceptions. Finally, we present tool prototypes built on top of SpotFlow to support software testing and code comprehension and we detail how SpotFlow runtime data can support novel empirical studies and datasets. SpotFlow is publicly available at https://github.com/andrehora/spotflow. Video: https://youtu.be/jhOv3nKz_u4.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Runtime environments**.

KEYWORDS

dynamic analysis, runtime monitoring, software testing, code comprehension, debugging, Python

ACM Reference Format:
Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639478.3640029

1 INTRODUCTION

Runtime (or dynamic) analysis is the ability to track what is happening during program execution [15]. Understanding the runtime behavioral aspects of a software system is fundamental for several software engineering tasks, such as testing, code comprehension, and debugging [3]. To this end, typically, one needs to instrument the system and collect data from its execution. Distinct data can be collected at runtime to be further analyzed, including executed lines, method calls, and execution time, to name a few [2].

Despite the importance of runtime analysis, few tools have been created and made public to support developers extracting information from software execution. For example, in a recent literature review about runtime monitoring, Rabiser *et al.* [12] found that most of the analyzed tools are not available (anymore) to the public.

In this paper, we propose SpotFlow, a tool to ease runtime analysis in Python (Section 2). SpotFlow executes and monitors a target Python program, collecting detailed information on method calls and states. For a more precise analysis, SpotFlow gathers data at the method-level for every method call, such as executed lines, argument values, return values, variable states, and thrown exceptions. SpotFlow can be used by practitioners and researchers working on the dynamic analysis of Python programs. SpotFlow is publicly available at https://github.com/andrehora/spotflow.

Finally, we discuss three practical applications of SpotFlow. *First*, we present PathSpotter,[1] a tool built on top of SpotFlow for computing and exploring tested paths[2] of Python methods (Section 3.1). We relied on PathSpotter to enhance the test suites of real-world systems, contributing with pull requests that were accepted and merged into popular projects, such as CPython, Rich, Jupyter Client, and Pylint. *Second*, on the top of SpotFlow, we built a prototype tool to visualize our runtime data (Section 3.2). *Third*, we detail how SpotFlow runtime data can support novel empirical studies and datasets (Section 3.3).

Novelty. The method call and state data collected by SpotFlow provides the basis for developing novel tools and applications, like PathSpotter. For instance, SpotFlow can directly detect *what classes, methods, test methods, or calls ran which lines.* This overcomes a limitation found in tracing tools [11], which typically work at the file-level and can only detect *what files ran which lines.*

Contributions. The contributions of this paper are twofold. First, we provide SpotFlow, a publicly available tool to ease runtime analysis in Python. Second, we discuss applications to support software testing, code comprehension, and novel empirical studies.

2 SPOTFLOW

2.1 Overview

SpotFlow runs and monitors a target Python program. The target program is defined by the user and can be one or more Python modules, classes, methods, or functions. SpotFlow collects method (and function) call and state data when monitoring a program. This is done to facilitate fine-grained runtime analysis, so we can precisely track the origin of runtime events. As we work at the method-level,

[1]https://github.com/andrehora/pathspotter
[2]https://andrehora.github.io/pathspotter/examples/report_html/gzip/index.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0502-1/24/04…$15.00
https://doi.org/10.1145/3639478.3640029

SpotFlow records runtime data, such as argument values, return values, variable values, thrown exceptions, and executed lines.

2.2 Domain Model

Figure 1 presents the domain model of SpotFlow. `MonitoredProgram` is a repository of monitored methods, which can be used to access all collected data. `MonitoredMethod` represents a monitored method. It has method calls and contains static information about the method/function, like name, full name, class name, file name, LOC, source code, etc. `MethodCall` represents a method call that happens at runtime and includes data about the caller, call stack, and executed lines. `CallState` holds the state of a method call, with information about argument states (`ArgState`), return states (`ReturnState`), thrown exceptions (`ExceptionState`), and local variable states (`VarStateHistory`). States know their runtime value, runtime type, and line number. Finally, `VarStateHistory` holds every state of a local variable in a method call. Notice that it is composed of variable states (`VarState`), representing the fact that a variable may change its value and can have multiple states over time.

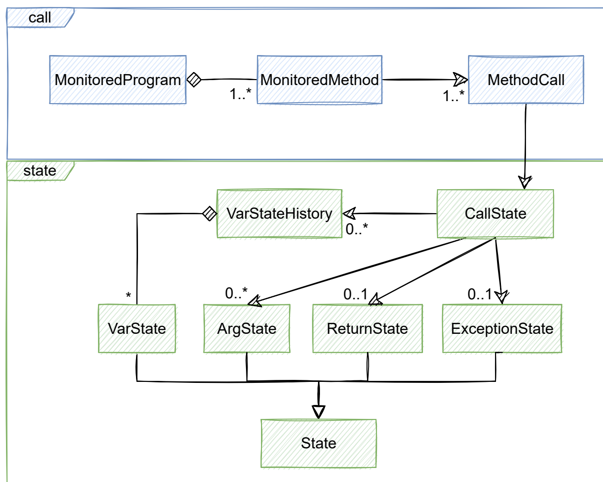

Figure 1: SpotFlow domain model.

2.3 Usage

SpotFlow can be run from the command line or programmatically via API. The running result of SpotFlow is a `MonitoredProgram` object, the entry point for the monitored data. First, we can install SpotFlow via pip:[3]

```
# Installing SpotFlow
$ pip install spotflow
```

We can use SpotFlow to collect data from the execution of any Python program. For example, to run `my_program.py`, we could originally use the following command-line:

```
# Running a Python program
$ python -m my_program
```

The same program can be run (and monitored) under SpotFlow with following command-line:

```
# Running a Python program + SpotFlow
$ python -m spotflow -t <target> my_program
```

The optional argument `-t` represents the target entity to be monitored. We can pass the full name of the target method (in the format *module.Class.method*) or a prefix to monitor multiple methods and classes. The final mandatory argument is the original command line, which is in this case *my_program*.

We can use SpotFlow to monitor the execution of test suites in both unittest and pytest frameworks. For example, to run a test `testX.py` under SpotFlow, the following change would be needed in the unittest command line:

```
# Running unittest
$ python -m unittest testX
# Running unittest + SpotFlow
$ python -m spotflow -t <target> unittest testX
```

2.4 Example

Suppose we have the target method `count_uppercase_words` (see Listing 1) and two test methods, as presented in Listing 2.

```
1 class StringParser:
2
3     def count_uppercase_words(self, text):
4         counter = 0
5         for word in text.split():
6             if word.isupper():
7                 counter += 1
8         return counter
```

Listing 1: Target method (parser.py).

```
1  class TestStringParser(unittest.TestCase):
2
3      def test_find_multiple_uppercase_words(self):
4          p = StringParser()
5          counter = p.count_uppercase_words("ABC DEF")
6          self.assertEqual(counter, 2)
7
8      def test_not_find_uppercase_word(self):
9          p = StringParser()
10         counter = p.count_uppercase_words("abc")
11         self.assertEqual(counter, 0)
```

Listing 2: Test suite (test_parser.py).

After running this test under SpotFlow, it produces the results summarized in Figure 2. The `MonitoredProgram` object holds the monitored methods, which is in this case only method `count_uppercase_words`. Notice that the monitored method `count_uppercase_words` has two calls, one from each test method.

The first call runs all lines of the monitored method, as we can check in `run_lines`. The state of the first call includes information about its argument, return, and variable states. The argument of the first call is the string "*ABC DEF*" and the return value is the int *2* that happens in line 8 of the monitored method. Note that the monitored method has two local variables: `counter` and `word`. The states of those variables over time are also recorded, for example, we can check that `counter` has the values *0*, *1*, and *2*, while `word` has the values "*ABC*" and "*DEF*" due to the text split.

Lastly, in the second call, we see that line 7 of the monitored method was not executed. In this call, the argument is "*abc*" and the return value is *0*. The local variable `counter` is always *0* (as it is not incremented), while `word` is "*abc*" (as the text is not split).

[3] https://pypi.org/project/spotflow

```
MonitoredProgram:
- monitored_methods: ['parser.StringParser.count_uppercase_words']

MonitoredMethod:
- name: 'count_uppercase_words'
- full_name: 'parser.StringParser.count_uppercase_words'
- class_name: 'StringParser'
- filename: 'parser.py'
- calls: [MethodCall 1, MethodCall 2]

MethodCall 1:
- caller: 'test_find_multiple_uppercase_words'
- run_lines: [3, 4, 5, 6, 7, 8]
- call_state: CallState 1

CallState 1:
- arg_states: [ArgState(name='text', value='ABC DEF']
- return_state: ReturnState(value=2, lineno=8)
- var_states:
  - counter: VarStateHistory([0, 1, 2])
  - word: VarStateHistory(['ABC', 'DEF'])

MethodCall 2:
- caller: 'test_not_find_uppercase_word'
- run_lines: [3, 4, 5, 6, 8]
- call_state: CallState 2

CallState 2:
- arg_states: [ArgState(name='text', value='abc']
- return_state: ReturnState(value=0, lineno=8)
- var_states:
  - counter: VarStateHistory([0])
  - word: VarStateHistory(['abc'])
```

Figure 2: Example of SpotFlow result objects.

2.5 Implementation Notes

SpotFlow is implemented with the support of the standard system's trace function `sys.settrace` [11]. This function is the basis for performing runtime analysis in Python, for instance, it is used to build Coverage.py, the *de facto* coverage tool for Python. The trace function allows for registering a hook that gets called at every executed line of code, function call, function return, and exception.

SpotFlow registers to the hook, monitors those events, and collects the domain model objects presented in Figure 1, such as `MethodCall` and `CallState`. Unfortunately, the trace function does not provide a simple way to collect those objects. That is, when a certain line of code is being executed, the trace function does not inform in which code entity the line is located, for example, in a method, class method, function, or local function. To overcome this limitation and find the proper data, SpotFlow performs the inspection of live objects on the current stack frame. For this purpose, we rely on the *inspect*[4] module, which provides functions to help get information about live objects, such as modules, classes, methods, functions, and frame objects.

2.6 Evaluation: Runtime Overhead

To evaluate the overhead added by SpotFlow, we execute the test suites of five popular Python libraries: json, ast, gzip, csv, and os. In total, those libraries have 797 tests. For each library, we report the average execution time (in seconds) over five runs. Table 1 presents the original execution time of each test suite and the execution time of SpotFlow in two settings: (1) the default, when `VarStateHistory` is not collected, and (2) the full, when all data is collected. In the default setting, the added overhead varies from +2.0x to +22.4x, while in the full setting, it varies from +4.1x to +64.1x. The overhead imposed by SpotFlow is not negligible, mainly in the full setting, however, it is in line with similar runtime tools [4, 7].

[4]https://docs.python.org/3/library/inspect.html

Table 1: Execution time of SpotFlow.

Project	#Tests	Execution time (seconds)		
		Test Suite	SpotFlow Default	SpotFlow Full
json	168	0.65	1.27 (+2.0x)	2.65 (+4.1x)
ast	139	0.61	2.24 (+3.7x)	3.49 (+5.7x)
gzip	61	0.25	1.75 (+7.0x)	2.69 (+10.7x)
csv	113	0.04	0.70 (+14.3x)	2.83 (+57.4x)
os	316	0.92	20.82 (+22.4x)	59.49 (+64.1x)

3 PRACTICAL APPLICATIONS

3.1 Software Testing

Due to the granularity level of analysis, SpotFlow can support the development of novel software testing tools. In this context, we developed PathSpotter, a tool for computing and exploring tested paths of Python methods. A tested path of a method represents a set of input values that will make the method behave in the same way, that is, execute the same lines of code. PathSpotter generates HTML reports for the whole project[5] and individual methods.[6] PathSpotter can be used to perform equivalence partitioning and boundary value analysis to support testing [1]. We relied on PathSpotter to improve the test suites of real-world projects. We successfully contributed with test improvement pull requests that were accepted and merged in highly relevant projects, such as CPython (PR 101378), Rich (PR 2786), Jupyter Client (PR 929), and Pylint (PR 8159).

As an example, Figure 3 presents the tested paths of the gzip method `flush`.[7] This method has a total of 25 calls and three tested paths. Figure 3(a) presents Path 1 (when the conditional in line 3 is true), which has 17/25 calls (68.0%). Figure 3(b) shows Path 2 (when the conditional in line 3 is false), which includes 7/25 calls (28.0%). Lastly, Figure 3(c) presents Path 3 with 1/25 (4.0%), when the ValueError exception is thrown in line 2.

3.2 Code Comprehension

Understanding the behavioral aspects is fundamental to improving code comprehension. For this purpose, visual solutions to learn programming have been proposed to aid developers in actually seeing the program state [8]. In this context, on the top of SpotFlow, we built a prototype tool to visualize our runtime data. This tool could be part of some visual solution to better understand the internal behaviors of a method. As an example (see Figure 4), we configured the tool to present: the executed lines of code (✅), the argument values (🟢), the return values (🔴), and the variable values over time (🟡). In this kind of solution, we can uncover every variable value, visually supporting code comprehension.

[5]Example: https://andrehora.github.io/pathspotter/examples/report_html/calendar
[6]Example: https://andrehora.github.io/pathspotter/examples/report_html/calendar/calendar.monthrange.html
[7]https://andrehora.github.io/pathspotter/examples/report_html/gzip/gzip.GzipFile.flush.html

```
1 def flush(self,zlib_mode=zlib.Z_SYNC_FLUSH):
2         self._check_not_closed()
3         if self.mode == WRITE:
4             # Ensure the compressor's buffer is flushed
5             self.fileobj.write(self.compress.flush(zlib_mode))
6             self.fileobj.flush()
```

(a) Path 1: 17/25 calls (conditional in line 3 is true).

```
1 def flush(self,zlib_mode=zlib.Z_SYNC_FLUSH):
2         self._check_not_closed()
3         if self.mode == WRITE:
4             # Ensure the compressor's buffer is flushed
5             self.fileobj.write(self.compress.flush(zlib_mode))
6             self.fileobj.flush()
```

(b) Path 2: 7/25 calls (conditional in line 3 is false).

```
1 def flush(self,zlib_mode=zlib.Z_SYNC_FLUSH):
2         self._check_not_closed()
3         if self.mode == WRITE:
4             # Ensure the compressor's buffer is flushed
5             self.fileobj.write(self.compress.flush(zlib_mode))
6             self.fileobj.flush()
```

(c) Path 3: 1/25 call (exception is thrown in line 2).

Figure 3: Tested paths of `gzip.GzipFile.flush`.

```
1 ⬜ def count_uppercase_words(text):    🟢 text='ABC DEF'
2 ✅     counter = 0                     🟡 counter=0
3 ✅     for word in text.split():       🟡 word='ABC' word='DEF'
4 ✅         if word.isupper():
5 ✅             counter += 1            🟡 counter=1 counter=2
6 ✅     return counter                  🔴 2
```

Figure 4: Calls and states of `count_uppercase_words`.

3.3 Empirical Studies and Datasets

1: Empirical Studies on Runtime Analysis. As a motivational example, we analyze with SpotFlow the test suite of the gzip Python library. We find that 31 gzip methods are executed 14,366 times. The most called method is: `_PaddedFile.read`[8] (5,432 calls). Among all calls, 10,865 return some value, while the others return void or throw an exception. From the 10,865 calls that return some value, 1,371 return boolean values. Digging a bit more, we detect that those returned boolean values are *true* in 1,300 calls (95%) and *false* in 71 ones (5%). We also find that 63 exceptions are thrown at runtime: `EOFError` (40), `ValueError` (11), `FileExistsError` (5), `TypeError` (3), `BadGzipFile` (3), and `UnsupportedOperation` (1). This kind of analysis exemplifies how we can explore the runtime data, for example, to understand common (and rare) testing scenarios [1].

2: Datasets with Runtime Metrics. SpotFlow can support the creation of novel datasets with a diversity of dynamic metrics. As an example, we present two datasets created with SpotFlow. To create them, we analyzed the test suites of 15 popular Python libraries.

•*Dataset 1: Variables Values at Runtime.* We extracted every variable name and their respective values at runtime. The dataset contains 1,234 distinct variables and a total of 133,169 distinct values. This kind of dataset can be used to gauge the quality of the tested data and to support the improvement of fake data generators [5].

•*Dataset 2: Mapping Between Test Cases and Application Methods.* We extracted every test method that executes at least one application method and mapped each test to their respective executed methods. The dataset contains 2,458 test methods and 42,218 executed application methods. In total, the application methods were executed 2,722,746 times by the tests. This kind of dataset has multiple applications for researchers, for example, to build coverage matrix [9], to support automated test generation [14], and to support code execution prediction with Large Language Models (LLMs) [10, 13].

> Dataset: https://doi.org/10.5281/zenodo.10015299.

[8] https://github.com/python/cpython/blob/c051d55/Lib/gzip.py#L88

4 RELATED WORK

Dynamic analysis is fundamental for several software engineering tasks, such as software testing, program comprehension, and debugging [3, 6, 8]. Unfortunately, few tools have been created and made public to support developers extracting information from software execution. Rabiser *et al.* [12] found that most monitoring tools are not publicly available. In Python, an exception is DynaPyt [4], a dynamic analysis framework that offers hooks into specific kinds of runtime events, such as function calls, writes of object attributes, and control flow decisions. DynaPyt does not have hooks that get called at every line of code as it focuses on AST constructors. Thus, unfortunately, we could not rely on DynaPyt to build tools that rely on the executed lines of code, like PathSpotter. In Python, there is also the native trace function `sys.settrace` [11]. However, its results only present executed lines at the file-level. In contrast, SpotFlow works at the method-level to track calls and create high-level objects, such as `MethodCall` and `CallState`.

5 CONCLUSION

In this paper, we proposed SpotFlow, a tool to ease the runtime analysis of Python programs. We discussed practical applications and presented tools, empirical studies, and datasets built with SpotFlow. As future work, we plan to build novel datasets to support empirical studies and tools to support software development.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.
[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to runtime verification. In *Lectures on Runtime Verification*. Springer, 1–33.
[3] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.
[4] 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.
[5] Faker: Faker is a Python package that generates fake data for you. October, 2023. https://github.com/joke2k/faker.
[6] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer* 23, 2 (2021), 255–284.
[7] 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.
[8] LearnableProgramming: Designing a programming system for understanding programs. October, 2023. http://worrydream.com/LearnableProgramming.

[9] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *International Conference on Software Engineering*. IEEE, 661–673.

[10] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code Execution with Pretrained Language Models. *arXiv preprint arXiv:2305.05383* (2023).

[11] Python sys.settrace. October, 2023. https://docs.python.org/3/library/sys.html.

[12] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software* 125 (2017), 309–321.

[13] Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement. 2023. Predicting Code Coverage without Execution. *arXiv preprint arXiv:2307.13383* (2023).

[14] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In *International Conference on Mining Software Repositories*. 299–303.

[15] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org