PyMOP: A Runtime Verification Tool for Python

Zhuohang Shen
Cornell University
Ithaca, NY, USA
zs435@cornell.edu

Mohammed Yaseen
Independent Researcher
Istanbul, Türkiye
moha.98.1900@gmail.com

Kevin Guan
Cornell University
Ithaca, NY, USA
kzg5@cornell.edu

Denini Silva
Federal University of Pernambuco
Recife, Pernambuco, Brazil
dgs@cin.ufpe.br

Marcelo d'Amorim
North Carolina State University
Raleigh, NC, USA
mdamori@ncsu.edu

Owolabi Legunsen
Cornell University
Ithaca, NY, USA
legunsen@cornell.edu

Abstract

Runtime verification (RV) now scales for testing in thousands of open-source Java projects, helping find hundreds of bugs by monitoring test executions against formal specifications (specs). The popular Python ecosystem could use such benefits. But, current Python RV tools are limited to a domain or spec logic, or they are slow. We present PyMOP, a generic, extensible, and more efficient Python RV tool. PyMOP supports five logics, implements five monitoring algorithms, ships with 81 specs, and supports three instrumentation strategies. On 48,090 unit tests in 839 GitHub projects, we find mainly that (i) PyMOP is up to **419.23x** faster than two recent dynamic analysis tools; (ii) Expensive instrumentation is a main cause of RV's runtime overhead for Python; and (iii) 84 of 156 bugs that PyMOP found were confirmed or fixed by developers. PyMOP's genericness and efficiency position it well as an excellent platform for the next advances on RV for Python. PyMOP is at https://github.com/SoftEngResearch/pymop and a video demo is at https://pymop.zhuohangshen.com/demo.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

runtime verification, software testing

ACM Reference Format:
Zhuohang Shen, Mohammed Yaseen, Kevin Guan, Denini Silva, Marcelo d'Amorim, and Owolabi Legunsen. 2026. PyMOP: A Runtime Verification Tool for Python. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3806424

1 Introduction

Runtime verification (RV) [21, 30, 54] monitors program executions against formal specifications (specs). RV has been used to mitigate failures and attacks in deployed software [13, 17] and to amplify the bug-finding ability of tests. RV of *passing* tests in thousands of Java projects against specs of correct usage of JDK API protocols helped find hundreds of bugs that testing alone missed [19, 33, 40].

Despite these advances, RV tools are still limited for Python, an increasingly important software ecosystem [59]. For example, PythonRV [50] works only for a fragment of Python syntax, and its overheads were not evaluated. LogScope [1] and PyContract [10] analyze logs offline; they do not work online, *e.g.*, during continuous integration (CI). Domain-specific Hat-RV [63] and VyPR2 [11, 27] only target embedded systems and performance monitoring of web services. DynaPyt [16] and DyLin [15] are dynamic analysis tools; their manually written "checkers" can be seen as RV monitors.

In sum, Python RV tools today have one or more of these limitations: (i) They are hard-wired to a spec logic, but no logic can express all specs [52] and users may want other logics [57]. (ii) Those that use logic, *e.g.*, linear temporal logic (LTL), can only monitor regular patterns; not context-free patterns [39]. (iii) Those that require manually writing monitors cannot use algorithms for synthesizing efficient monitors from specs [22, 23, 38, 39, 53, 55, 56, 58]. (iv) None uses the parametric monitoring algorithms [51] that helped RV scale for testing in Java. (v) None was evaluated at scale with dozens of specs and hundreds of projects, as was done for Java [19, 26, 33, 34]. (vi) None supports online *and* offline monitoring.

We present PyMOP, a generic, extensible, and more efficient Python RV tool that addresses these limitations. PyMOP is *generic* in four ways. First, it is not hard-wired to a logic, and users can extend it with plugins for more logics. PyMOP ships with five logic plugins (with more planned in the future): past- and future-time LTL, extended regular expressions (ERE), finite-state machines (FSM), and context-free grammars (CFG). Second, PyMOP does not require an external domain-specific language (DSL); users can write specs directly in Python via an embedded API (*i.e.* an internal DSL), or in a JSON-like syntax. Third, PyMOP has an API to support different parametric monitoring algorithms (currently five), with documentation in our artifact on how to extend it. PyMOP also supports three instrumentation strategies. Lastly, PyMOP is not domain specific—its current specs are about APIs in Python itself and also in widely-used libraries, *e.g.*, Tensorflow (machine learning), SciPy (scientific computing), and Pandas (data analysis).

PyMOP's *efficiency* comes from two sources: it (i) automatically synthesizes efficient monitors from specs; and (ii) uses parametric trace-slicing based monitoring algorithms [51]. PyMOP is the first Python instance of Monitoring-Oriented Programming

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3806424

```
{ "Description": "Detects TOCTOU...",
  "Variables": {"checked_files": "set"},
  "Formalism": "fsm",
  "Formula": "s0 [use -> s1, check -> s2] s1 [use -> s1, check
       -> s2] s2 [use -> s3, check -> s2] s3 [use -> s3, check
       -> s2] alias Violation = s3",
  "Creation_Events": ["check"],
  "Events": {
    "After": {
      "check":[["os","access"]],"use":[["builtins","open"]]}},
  "Event_Actions": {
    "After": {
      "check": "self.checked_files.add(f)", "use": "return file
           in self.checked_files" }},
  "Handlers": {"Violation": "Security threat! ..."  } }
```

Figure 1: TOCTOU spec in PyMOP's JSON-like frontend.

(MOP) [7], which scales well for simultaneous monitoring of multiple specs [37]. PyMOP could play a similar role for Python to what JavaMOP [29] played for Java; JavaMOP has been widely used for decades [4–6, 14, 19, 20, 25, 28, 29, 32, 35, 38, 42, 43].

Our evaluation is the largest for Python RV so far; it uses (i) 48,090 unit tests in 839 projects from many domains: machine learning, data processing, infrastructure-as-code, blockchains, etc.; (ii) three tools: PyMOP, DyLin, and DynaPyt; and (iii) 81 specs that we write about correct API usage in Python and widely-used libraries. For context, the prior largest real-world evaluation of RV for Python used 11 projects and 15 specs [15]. Without monitoring third-party libraries, PyMOP is up to 356.23x faster than state-of-the-art (SoTA) DynaPyt and DyLin. But, on average, PyMOP is 26.07x slower than running tests without RV and consumes 5.67x more memory. When monitoring libraries, PyMOP is up to 419.23x faster than DynaPyt (DyLin does not support libraries), while being 180.39x slower and using 8.69x more memory than running tests. So, future work is needed to speed up Python RV. Separately, PyMOP finds many violations that the SoTA miss. We reported 156 bugs that PyMOP helped us find; developers have so far fixed or confirmed 84. PyMOP and our data are at https://github.com/SoftEngResearch/pymop.

2 Illustrative Example

TOCTOU Spec. "Time-Of-Check to Time-Of-Use" (TOCTOU) vulnerabilities [41] occur when attackers can exploit the time between checking permissions on file `f` and using `f`, *e.g.*, by replacing `f` or changing `f`'s contents or permissions [3, 61]. PythonDocs describe TOCTOU [49]: *"Using access() to check if a user is authorized to e.g. open a file before actually doing so using open() creates a security hole"*. Figure 1 shows PyMOP's TOCTOU spec, using PyMOP's JSON-like syntax (users can also write specs directly in Python). That spec defines a FSM monitor. Using that FSM, PyMOP monitors all files that are used as arguments in `os.access()` calls and raises a security violation if a subsequent `open()` call uses one such file. In Figure 1, `Description` is a summary, `Variables` is the set `checked_files` of files tracked at runtime, `Formalism` is the spec logic (FSM in this case), `Formula` is the FSM in text form (Figure 2 visualizes it), and `Creation_Events` means the `check` event should trigger monitor synthesis at runtime. `Events` maps the `check` event to `os.access()` calls and the `use` event to `builtins.open()` calls.

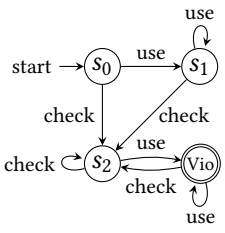

Figure 2: TOCTOU's FSM.

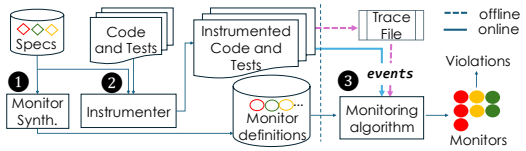

Figure 4: PyMOP's Architecture.

`Event_Actions` tell monitors what to do when events occur (*e.g.*, add `f` to `checked_files` on `check` events). Finally, `Handler` code is run if the FSM reaches a violating state; it can be any, *e.g.* error recovery, code. But, for testing, we print a message. A complete description of the spec syntax is available in our artifact.

TOCTOU violations found by PyMOP. We reported three TOCTOU violations that PyMOP helped us find; one was accepted, and the other two are pending. The accepted violation was in `mycli`[12], a popular command-line MySQL interface with 680 forks and 11.9K stars. That violation was in a function that reads passwords from a file. Figure 3 simplifies the vulnerable code (lines 1–5, which yield the violating trace `os.access(f)`⇝`open(f)`) and our fix (lines 6–10). Our fix, which was merged by `mycli` developers, replaces explicit access checks with more secure exception handling [3].

```
1 # Vulnerable version:
2 if os.path.isfile(password_file) and
3   os.access(password_file, os.R_OK):
4     with open(password_file) as fp:
5         return fp.readline().strip()
6 # Fixed version:
7 try:
8     with open(password_file) as fp:
9         return fp.readline().strip()
10 except FileNotFoundError: ...
```

Figure 3: A TOCTOU violation [12].

Verifying behavioral properties like TOCTOU statically can be challenging; one violation [44] involves check and use across functions. Interprocedural static analysis would be needed to verify TOCTOU here. Specs with multiple object types as parameters, plus the need to reason about library code would make static analysis slower [60].

3 PyMOP

Architecture. Figure 4 shows PyMOP's architecture, supporting *both* offline and online RV. Offline RV [1, 9, 10] logs signaled events for postmortem analysis. Online RV monitors (but does not log) events as they are signaled. PyMOP has three main components: Monitor Synthesizer, Instrumenter, and Monitoring Engine. The first two are identical for offline and online RV. Monitor Synthesizer (❶) uses logic plugins to compile specs (diamonds) into code (circles) from which the Monitoring Engine (❸) synthesizes monitors (filled circles) at runtime. Instrumenter (❷) rewrites the code and tests to signal events at runtime. Monitoring Engine (❸) synthesizes, dispatches events to, and garbage collects monitors. By separating monitor code generation from monitor synthesis, PyMOP performs potentially time-consuming monitor code synthesis once, reducing runtime overheads and allowing the Monitoring Engine to focus on synthesizing monitors (and garbage-collecting them) at runtime. PyMOP's modularity enables extensibility.

Implementation. We briefly discuss parts of how PyMOP works. ❶ *Monitor Synthesizer*. PyMOP invokes JavaMOP's mature and well-tested monitor-synthesis plugins for ERE, FSM, past- and future-time LTL specs. Doing so sped up our PyMOP development. But, we implement a new CFG monitor-synthesis plugin in Python, showing how to add native logic plugins. All plugins produce monitor

```
{ "TOCTOU_FileOpen": {
  "last_event: vulnerable_open, \
   message: Spec - TOCTOU_FileOpen : Security threat! Using
        access()... \
   file_name: path/to/project/file.py, \
   line_num: 42": {
    "count": 3,
    "test": [
      "tests/test_example.py::test_case_1",
      "tests/test_example.py::test_case_2"
    ] }}}
```

Figure 5: PyMOP's JSON report for a TOCTOU spec violation.

templates (circles in Figure 4) from which runtime monitors are synthesized at runtime. For ERE, FSM, and LTL, that template defines an FSM. For CFG, that template defines a push-down automaton [39].

❷ *Instrumenter.* PyMOP supports three instrumentation strategies: (i) Python-level monkey patching only [24], (ii) Python-level monkey patching plus C-level monkey patching via the curses function in `forbiddenfruit` [8], and (iii) Python-level monkey patching plus AST transformation [48]. Using these strategies, PyMOP rewrites the code under test (CUT), libraries, and the Python runtime to signal events right before or after they occur at runtime. Our appendix [45] discusses in greater detail the trade offs in capability and cost among these three strategies. Strategy (iii) achieves the best balance in practice: while alternative strategies can be faster, they miss many events (particularly for methods that are implemented in CPython). In contrast, AST-based instrumentation provides more complete event coverage with lower speed, and we therefore recommend it as the default in PyMOP.

❸ *Monitoring Algorithms.* RV specs are often parametric, meaning that a behavioral property must hold *for all* sets of related objects. For example, a parametric PyMOP spec is `PyDocs_MustSortBeforeGroupBy`: for each list `L`, if `itertools.groupby(L,...)` is called, then `L` must have been previously sorted. This spec has two parameters: `L` and `groupby`'s `iterator`. So, RV must check that *all* such (`L`, `iterator`) pairs created at runtime are correctly monitored. Trace-slicing algorithms [51] were proposed for correct and efficient monitoring of parametric specs, and they are at the heart of why the MOP approach to RV scales. Intuitively, for each spec, a trace-slicing algorithm decomposes the program trace into subtraces per parameter sets. Then those parameters are "forgotten" and each sub-trace is monitored in a non-parametric way.

We implement five trace-slicing algorithms in PyMOP: $\mathbb{A}\langle X\rangle$, $\mathbb{B}\langle X\rangle$, $\mathbb{C}\langle X\rangle$, $\mathbb{C}^+\langle X\rangle$, and $\mathbb{D}\langle X\rangle$ (see [51] for details). $\mathbb{A}\langle X\rangle$ is offline RV and only checks one spec at a time. The others are for online RV; $\mathbb{C}\langle X\rangle$, $\mathbb{C}^+\langle X\rangle$, and $\mathbb{D}\langle X\rangle$ are optimized versions of $\mathbb{B}\langle X\rangle$ and we implement them all because they make different time and space trade offs that can make them faster or slower than one another in different projects. The default monitoring algorithm in PyMOP is $\mathbb{D}\langle X\rangle$: it performed best most frequently in our early experiments.

Specifications. PyMOP ships with 81 specs that use all its logic plugins. We manually write these 81 API-level specs of the Python language (*Python specs*) and of widely-used Python libraries (*Library specs*). To do so, we follow Lee et al. [37]'s procedure for obtaining API specs. We first search API docs for hints on API usage constraints captured in sentences with *"should"*, *"only"*, *"must"*, *"note"*, etc. (Our artifacts contain the complete list of keywords.) We then manually check if specs should be written for these sentences. Of 81 specs that we write, 61 are Python specs and 20 are library specs for `nltk`, `requests`, `tensorflow`, `flask`, `tornado`, and `scipy` (click links). 11 of these 81 specs are from DyLin; we plan to translate more in the future. Also, others can add specs using PyMOP's logic syntax in Python, or its JSON-like syntax.

Outputs. PyMOP generates a report containing all unique violations and the file path, line number, frequency, last triggering event and triggering test cases for each unique violation. Users can choose among report formats: plain text, JSON, or a JSON file. The default output format is JSON, *e.g.*, Figure 5 for a TOCTOU violation. For each spec, violations are grouped by program location; each entry records the last triggering event (`last_event`), an optional diagnostic message (`message`), the location (`file_name` and `line_num`), and metadata including the number of occurrences (`count`) and triggering test cases (`test`).

4 Installation and Usage

Installation. Users can install PyMOP via Docker or by directly installing PyMOP on a host machine using `pip`:

```
# Option 1: Using Docker
$ docker run -it softengresearch/pymop
# Option 2: Installing on host machine
$ pip install . # Run in PyMOP root directory
```

Usage. To use PyMOP, users must first set the `PYTHONPATH` environment variable to PyMOP's library. Then, the `PYMOP_SPEC_FOLDER` option must be set (via an environment variable or a configuration file, *e.g.*, `.pymop_env`) to the directory containing all specs to monitor. Finally, the user runs their program (we run tests).

```
# Enable PyMOP:
$ export PYTHONPATH=~/pymop/pythonmop/pymop-startup-helper
# Set specs folder via an environment variable:
$ export PYMOP_SPEC_FOLDER=~/pymop/specs-new
$ pytest tests # Run tests
```

Options. PyMOP provides many options for customizing its behavior. Detailed descriptions of these options are in our artifact.

5 Evaluation

Evaluated Projects. We use 839 projects where tests pass with and without PyMOP (for fair comparison). We obtain these projects in two steps. First, we use GitHub's REST API to find 631 projects whose manifest files mention libraries that we have specs for. Then, we use GitHub's Search to find 208 more spec-related projects that use `pytest`. Table 1 shows summary statistics about all 839 projects. The minimum number of tests is one, since prior work showed that RV incurs high overhead even in projects with few tests [19]. Some minimum values for other metrics may appear small (*e.g.*, projects with only two commits). We verified that these values are correct and chose to keep them, as PyMOP found violations in them.

Running Experiments. We run all experiments on a server with AMD® EPYC® 7763 64-Core Processor with 4 virtual CPUs and 15GB of RAM, and Ubuntu 24.04.2 LTS, using Docker.

5.1 Results

Comparing PyMOP with the SoTA. We compare PyMOP with SoTA DynaPyt and DyLin on 483 of 839 projects (DyLin and DynaPyt fail on the other 356), using 26 specs supported by all three

Table 1: Statistics on 839 evaluated projects: no. of tests (#Tests), time w/o RV in seconds (time), size (SLOC), no. of commits (#Sha), age in years (Age), no. of GitHub stars (⋆), no. of events (*evt*) and monitors (*mon*) while running tests with RV. "n/a" are meaningless sums.

	#Tests	time	SLOC	#Sha	Age	⋆	*evt*	*mon*
Avg	57.3	2.5	57,820.2	337.9	6.2	326.5	3.0×10^6	1,297.2
Med	11	0.6	2,417	95	5.8	19	2,306	107
Min	1	0.2	54	2	0.6	0	17	19
Max	3,525	133	8.4×10^6	31,925	16.4	34,636	6.1×10^8	1.4×10^5
Sum	48,090	2,140	4.9×10^7	n/a	n/a	n/a	2.5×10^9	1.1×10^6

tools (11 from DyLin and 15 from PyMOP). We use DynaPyt and DyLin as baselines, but they were only previously evaluated only on CUT [15, 16]. With input from their authors, we add support in DynaPyt to also enable evaluation on third-party libraries. DyLin cannot monitor libraries.

Table 2 compares the absolute time summed across all 483 projects (in hours), total violations, and average memory usage for tests without RV, PyMOP (with and without libraries), DynaPyt (with and without libraries), and DyLin (without libraries). There, the first three rows show instrumentation-only, monitoring-only, and end-to-end times in hours; the later is the sum of the former two.

PyMOP has the lowest instrumentation-only, monitoring-only, and end-to-end times without libraries, and also with libraries. When not monitoring libraries, PyMOP is up to 356.23x faster than DynaPyt (avg: 3.74x), up to 352.79x faster than DyLin (avg: 3.63x). When monitoring libraries, PyMOP is up to 419.23x faster than DynaPyt (avg: 163.26x). Also, PyMOP with and without library monitoring consume slightly more memory than DynaPyt and DyLin, suggesting that PyMOP's significant time improvement over SoTA comes with a small additional memory overhead. In all three tools, instrumentation times are a main cause of RV overhead. So, future work should investigate faster instrumentation for Python.

When not monitoring libraries, PyMOP finds 11 and 12 more violations than DynaPyt and DyLin, respectively, due to analysis simplifications in DynaPyt and DyLin that trade safety for performance. PyMOP seemingly misses two violations reported by DynaPyt and DyLin, but those are false positives caused by bugs in DynaPyt and DyLin. When monitoring libraries, PyMOP finds 87 more violations than DynaPyt due to said simplifications and bugs. But, PyMOP misses nine violations in code that runs pre-importation, before PyMOP gets a chance to instrument libraries (which happens after libraries are imported). Our future plans include the elimination of these violations missed by PyMOP in third-party libraries (not in the CUT). We reported the two false positives observed in the CUT to the DynaPyt and DyLin authors [46, 47].

PyMOP vs. running tests without RV. We measure the time and memory usage when running tests with and without PyMOP using all 81 specs and default options: instrumentation via `ast` and monitoring algorithm $\mathbb{D}\langle X\rangle$. Across all 839 projects, PyMOP is on average 26.07x slower than without RV (max: 2,217.8x) when not monitoring libraries. In absolute terms, PyMOP slows tests down by 47.29 seconds on average, and as much as 2.18 hours. Also, on average, PyMOP consumes 5.67x (or 0.21 GB) more memory than without RV, with a max of 28.07x (or 1.52 GB). When CUT and libraries code are monitored, PyMOP's runtime overhead is, on average, 180.39x (or 5.72 minutes), and up to 17,977.83x (or 5.25 hours). Also, PyMOP uses 8.69x (or 0.39 GB) more memory than without RV, and up to 184.17x (or 11.71 GB) more. The higher memory overhead is primarily due to maintaining large numbers of monitor instances and internal data structures, but could be mitigated through more efficient data structures [28]. Table 2 shows that these runtime overheads are lower than those of the SoTA. But, prior work on speeding up JavaMOP could be adapted in the future to make PyMOP faster [18, 20, 31, 36, 62].

Table 2: Sums across all 483 projects of end-to-end time (in hours), unique violations, and average memory usage (in MB) for tests without RV, PyMOP, DynaPyt, and DyLin.

	w/o RV	PyMOP (no libs)	DynaPyt (no libs)	DyLin (no libs)	PyMOP (w/ libs)	DynaPyt (w/ libs)
Instr.	-	0.43	1.33	1.32	1.42	263.53
Monitoring	-	0.76	3.37	2.58	2.78	11.42
End-to-End	0.24	1.19	4.7	3.9	4.2	274.94
Violations	-	37	28	27	128	50
Memory	45.11	178.41	56.43	57.78	212.35	76.52

Bugs Found. We have so far inspected 271 violations in 124 projects. 156 are true bugs, 100 are false alarms (*i.e.*, the violated spec is buggy), and 15 are not fixable (*i.e.*, the repository is no longer public). These false alarm rates are so far lower than those reported for Java [33, 34] and we will improve our specs to reduce them. We also plan to investigate developer-in-the-loop spec refinement and explore automated filtering of false alarms. We reported all 156 true bugs to developers, 84 were confirmed or fixed, nine were rejected, and 63 are pending. Among the 84 confirmed or fixed bugs, 13 involve improper resource cleanup, 28 are performance-related API misuses related to inefficient membership checks or string concatenation, and 12 violate API call-ordering constraints. The remaining 31 bugs reflect violations of API documentation recommendations, including deterministic test seeding, input normalization, and edge case handling in built-in APIs such as `all` and `any`.

6 Limitations and Future Work

We plan to evaluate PyMOP on deployed programs, add monitor-synthesis plugins for other logics, *e.g.*, [2, 38], and add support for instrumenting Python type checkers implemented in C and library-level dictionaries. Also, future work on evolution-aware RV techniques for Python could further speed up PyMOP by focusing on code affected by changes during CI [36].

7 Conclusion

PyMOP is the first MOP instance for Python. It is *generic*, supporting five monitoring algorithms, three instrumentation strategies, and five spec logics (with 81 specs written), and features to add more of these. PyMOP is also more *efficient* than the SoTA, as our large-scale evaluation with 839 projects shows. PyMOP has so far helped us find 156 bugs during testing of open-source projects.

Acknowledgments. We thank the anonymous reviewers for their feedback. This work is partially supported by the United States NSF under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2319472, and CCF-2349961; and CNPq under Grant No. 140220/2022-4.

References

[1] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret Smith. 2010. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication* 7, 11.

[2] David Basin, Felix Klaedtke, and Eugen Zălinescu. 2017. Runtime verification of temporal properties over out-of-order data streams. In *Computer Aided Verification*.

[3] Matt Bishop. 2003. Program Security. *Computer Security: Art and Science*.

[4] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2004. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods*.

[5] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2006. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *Runtime Verification*.

[6] Feng Chen and Grigore Roşu. 2007. MOP: An efficient and generic runtime verification framework. In *Object-oriented Programming, Systems, Languages, and Applications*.

[7] Feng Chen and Grigore Roşu. 2003. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification*.

[8] Lincoln Clarete. 2026. Forbidden fruit. https://github.com/clarete/forbiddenfruit.

[9] Christian Colombo, Gordon J Pace, and Patrick Abela. 2009. *Offline runtime verification with real-time properties: A case study*. Technical Report. University of Malta.

[10] Dennis Dams, Klaus Havelund, and Sean Kauffman. 2022. A Python library for trace analysis. In *Runtime Verification*.

[11] Joshua Heneage Dawes and Giles Reger. 2019. Specification of temporal properties of functions for runtime verification. In *Symposium on Applied Computing*.

[12] dbcli 2026. MyCLI. https://github.com/dbcli/mycli/.

[13] Daniel Bristot de Oliveira. 2021. Efficient runtime verification for the Linux kernel. https://research.redhat.com/blog/article/efficient-runtime-verification-for-the-linux-kernel.

[14] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime monitoring with union-find structures. In *Tools and Algorithms for the Construction and Analysis of Systems*.

[15] Aryaz Eghbali, Felix Burk, and Michael Pradel. 2025. DyLin: A dynamic linter for Python. In *Foundations of Software Engineering*.

[16] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A dynamic analysis framework for Python. In *Foundations of Software Engineering*.

[17] GrammaTech 2026. ARTCAT: Autonomic response to cyber-attack. https://www.grammatech.com/artcat-autonomic-response-to-cyber-attack.

[18] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster explicit-trace monitoring-oriented programming for runtime verification of software tests. In *Object-oriented Programming, Systems, Languages, and Applications*.

[19] Kevin Guan and Owolabi Legunsen. 2024. An in-depth study of runtime verification overheads during software testing. In *International Symposium on Software Testing and Analysis*.

[20] Kevin Guan and Owolabi Legunsen. 2025. Instrumentation-driven evolution-aware runtime verification. In *International Conference on Software Engineering*.

[21] Klaus Havelund and Grigore Roşu. 2001. Monitoring programs using rewriting. In *Automated Software Engineering*.

[22] Klaus Havelund and Grigore Roşu. 2002. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*.

[23] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online monitoring of metric temporal logic. In *Runtime Verification*.

[24] John Hunt. 2023. Monkey patching. In *A Beginners Guide to Python 3 Programming*.

[25] Soha Hussein, Patrick Meredith, and Grigore Roşu. 2012. Security-policy monitoring and enforcement with JavaMOP. In *Programming Languages and Analysis for Security*.

[26] Omar Javed and Walter Binder. 2018. Large-scale evaluation of the efficiency of runtime-verification tools in the wild. In *Asia-Pacific Software Engineering Conference*.

[27] Omar Javed, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. 2020. PerfCI: A toolchain for automated performance testing during continuous integration of Python projects. In *Automated Software Engineering, Demo Track*.

[28] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation*.

[29] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *International Conference on Software Engineering, Demo Track*.

[30] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *Euromicro Conference on Real-Time Systems*.

[31] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2025. Faster runtime verification during testing via feedback-guided selective monitoring. In *Automated Software Engineering*.

[32] Choonghwan Lee, Feng Chen, and Grigore Roşu. 2011. Mining parametric specifications. In *International Conference on Software Engineering*.

[33] Owolabi Legunsen, Nader Al Awar, Xinyue Xu, Wajih Ul Hassan, Grigore Roşu, and Darko Marinov. 2019. How effective are existing Java API specifications for finding bugs during runtime verification? *Automated Software Engineering Journal* 26, 4.

[34] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *Automated Software Engineering*.

[35] Owolabi Legunsen, Darko Marinov, and Grigore Roşu. 2015. Evolution-aware monitoring-oriented programming. In *International Conference on Software Engineering, NIER Track*.

[36] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for evolution-aware runtime verification. In *International Conference on Software Testing, Verification and Validation*.

[37] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Runtime Verification*.

[38] Patrick Meredith and Grigore Roşu. 2013. Efficient parametric runtime verification with deterministic string rewriting. In *Automated Software Engineering*.

[39] Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. 2008. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering*.

[40] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing runtime verification violations. In *International Conference on Software Testing, Verification and Validation*.

[41] MITRE. 2024. CWE-367: Time-of-check time-of-use (TOCTOU) race condition. https://cwe.mitre.org/data/definitions/367.html.

[42] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2010. Monitor optimization via stutter-equivalent loop transformation. In *Object-oriented Programming, Systems, Languages, and Applications*.

[43] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2013. Optimizing monitoring of finite state properties through monitor compaction. In *International Symposium on Software Testing and Analysis*.

[44] PyMOP Team 2025. LiveDiffer bug PR. https://github.com/manthanmtg/live_differ/pull/3/.

[45] PyMOP Team 2026. Appendix to this paper. https://github.com/SoftEngResearch/pymop/blob/main/appendix.pdf.

[46] PyMOP Team 2026. DyLin issue #6. https://github.com/sola-st/DyLin/issues/6.

[47] PyMOP Team 2026. DyLin issue #9. https://github.com/sola-st/DyLin/issues/9.

[48] Python Team 2026. Python standard library documentation for AST. https://docs.python.org/3/library/ast.html.

[49] Python Team 2026. Python standard library documentation for OS. https://docs.python.org/3.10/library/os.html#os.access.

[50] Adam Renberg. 2014. *Test-inspired runtime verification: Using a unit test-like specification syntax for runtime verification*. Master's thesis. KTH, Sweden.

[51] Grigore Roşu and Feng Chen. 2012. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science* 8.

[52] Grigore Roşu. 2012. On safety properties and their monitoring. *Scientific Annals of Computer Science* 22, 2.

[53] Grigore Roşu and Saddek Bensalem. 2006. Allen linear (interval) temporal logic – Translation to LTL and monitor synthesis. In *Computer Aided Verification*.

[54] Fred B. Schneider. 2000. Enforceable security policies. *Transactions on Information and System Security* 3, 1.

[55] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. 2019. A formally verified monitor for metric first-order temporal logic. In *Runtime Verification*.

[56] Koushik Sen, Grigore Roşu, and Gul Agha. 2003. Generating optimal linear temporal logic monitors by coinduction. In *Advances in Computing Science – ASIAN*.

[57] Leopoldo Teixeira, Breno Miranda, Henrique Rebêlo, and Marcelo d'Amorim. 2021. Demystifying the challenges of formally specifying API properties for runtime verification. In *International Conference on Software Testing, Verification and Validation*.

[58] Prasanna Thati and Grigore Roşu. 2004. Monitoring algorithms for metric temporal logic specifications. In *Runtime Verification*.

[59] TIOBE Software BV 2025. TIOBE Index. https://www.tiobe.com/tiobe-index.

[60] Adriano Torres, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, Marcelo d'Amorim, Owolabi Legunsen, Eric Bodden, and Edna Dias Canedo. 2023. Runtime verification of crypto APIs: An empirical study. *Transactions on Software Engineering*.

[61] Jinpeng Wei and Calton Pu. 2005. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *File and Storage Technologies*, Vol. 5.

[62] Ayaka Yorihiro, Pengyue Jiang, Valeria Marqués, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven plugin for evolution-aware runtime verification. In *Runtime Verification*.

[63] Wanjin Zhou, Feifei Hu, and Junyan Ma. 2022. Improving flexibility in embedded system runtime verification with python. In *International Symposium on Software Reliability Engineering Workshop*.