A Python Library for Trace Analysis

Dennis Dams[1], Klaus Havelund[2], and Sean Kauffman[3]

[1] ESI (TNO), The Netherlands
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA
[3] Aalborg University, Denmark

Abstract. We present a Python library for trace analysis named PyContract. PyContract is a shallow internal DSL, in contrast to many trace analysis tools that implement external or deep internal DSLs. The library has been used in a project for analysis of logs from NASA's Europa Clipper mission. We describe our design choices, explain the API via examples, and present an experiment comparing PyContract against other state-of-the-art tools from the research and industrial communities.

1 Introduction

Runtime Verification (RV) is an approach to checking that a system under execution does the right thing, complementary to testing and static verification. Numerous RV systems have been developed over time. Many of these systems offer so-called *external* domain-specific languages (DSLs) [19, 2, 9, 4, 7, 22, 5, 18, 11, 1]. These parse a specification and synthesize a runtime monitor from the specification. *Internal* (or embedded) DSLs, on the other hand, extend an existing host programming language, usually as a library.

There are two kinds of internal DSLs: deep and shallow [14]. In a *deep* internal DSL, data structures in the host language are used to represent DSL constructs in an explicit manner, e.g., as an Abstract Syntax Tree (AST), which can then be processed by writing either an interpreter or a compiler for execution. Some examples are [15, 26]. A *shallow* internal DSL includes the constructs of the host language as part of the DSL, using the host language's native runtime system to execute them. This is how most programming language libraries are implemented. Shallow Scala-internal DSLs developed by the authors, and their use, are described in [3, 16, 17, 20].

In this work we present a shallow Python-internal DSL for RV. Our motivation stems from our experiences in infusing monitoring technology into practice. The main reason for implementing a monitoring library in Python is the language's popularity [28]. The most recent version of Python provides pattern matching, which we demonstrate is useful for writing monitors. We believe that an internal DSL increases *adoption*. For example, in our experience with applying the CommaSuite tool [8] in industrial contexts, we have observed that having to learn a new specification language is sometimes experienced by potential users as a barrier to using the tool. An internal DSL is "just" another library in a familiar language, and modern programmers commonly use many libraries [24]. They

can continue to use their favorite development tools (such as IDEs) and other libraries for the host language. The chance of adoption is furthermore increased due to the fact that shallow internal DSLs offer full *expressiveness*, since the host language can be leveraged for complex calculations. This is an essential design point to expand PyContract's possible uses. This has been demonstrated by its application in the analysis of telemetry logs from testing the Europa Clipper mission [12] flight computer.

Furthermore, *design and implementation* are much easier for internal DSLs than for external DSLs, because the syntax and semantics of the host language are used. A noteworthy aspect of our solution is its small implementation. For the same reason, *maintenance* requires less effort. The effort to respond to feature requests can be reduced due to the availability of language constructs in the host language. E.g., the development of the external CommaSuite DSL has involved regularly occurring user requests for more "richness" in the language such as dictionaries (hash maps) as an additional data type and a notation for namespaces. In an external DSL, adding such features is made more challenging by the need to handle all aspects of the language interpretation. A potential disadvantage of shallow internal DSLs (when compared to external DSLs and deep internal DSLs) is *analyzability*. However, Python supports powerful meta-programming features allowing a program to inspect its own AST.

We support monitoring of events that carry data, allowing specification of the relationship between data in events arriving at different time points (i.e. we support first-order temporal properties). PyContract implements a form of *slicing (indexing)* [22, 26] for optimizing monitoring events with data, limiting the search when an event is submitted to the monitor. We use an *automaton* flavored language rather than temporal logic for the specification of monitors as we find automata to be more flexible. Our DSL resembles a cross between Extended Finite State Machines (EFSMs) [6], such as Quantified Event Automata (QEA) [26], and rule-based programming, such as RuleR [4]. It most closely resembles the Scala DSL Daut [16, 10].

We support *two flavors of states*. In *next*-states, the next event must match a transition. This allows the definition of state machines as found in standard textbooks on finite automata [27]. In *skip*-states, events may be skipped until a transition matches. This can be used for state machines in the style of SysML [29] and UML [31]. We support *skip*-states as the default and treat all states as implicitly accepting unless marked as rejecting. In our experience, this provides a flexible way to write concise monitors.

We support *visualization* of monitors to improve user comprehension. Visualization is an efficient way to communicate the meaning of a specification [30]. We use a format based on standard EFSM displays that leverages user familiarity with such diagrams.

2 The PyContract Library

PyContract is an internal Python DSL for writing event monitors. It is inspired by rule-based programming [4, 17] in that the memory of a monitor is a set of facts, where a fact is a named data record. Furthermore, facts, like states in state machines, can have transitions which, upon triggering, can generate other facts. As in EFSMs one can also define variables, local to a monitor, to which the transitions can refer in conditions and in actions. Finally, since the DSL is a Python library, one can write arbitrary Python code as part of the monitors. PyContract is inspired by the Scala DSL Daut [16, 10] and is developed for Python 3.10 that supports pattern matching [25]. We use pattern matching extensively for defining transition functions. The general approach is to define a monitor as a sub-class of the `Monitor` class, create an instance of it, and then feed it with events. Events are fed, one by one, using the `evaluate(event: Event)` method and, in the case of a finite sequence of observations, a call of the `end()` method signals the monitor that the sequence has ended, at which point any outstanding obligations that have not been satisfied (expected events that did not occur) will be reported as errors. PyContract is available under the Apache 2.0 open-source license at [23]. In the following we shall illustrate how to write monitors with two examples.

2.1 Example 1

Consider a sequence of events, where each event indicates the acquisition or the release of a lock by a thread. PyContract can monitor events of any kind: numbers, strings, dictionaries (maps), objects of user defined classes, etc. We shall here assume the definition of two such event classes `Acquire` and `Release` defined as data classes[1], each taking a thread and a lock as argument, allowing the construction of objects such as `Acquire(thread, lock)` and `Release(thread, lock)`, and performing pattern matching over these.

The monitor we shall present is a "kitchen sink" example of features, and implements property P_1, consisting of five sub-properties:

$P_{1.1}$ A thread acquiring a lock must eventually release it.
$P_{1.2}$ While a lock is acquired by a thread it cannot be acquired by any thread.
$P_{1.3}$ A thread can only release a lock if it has acquired it, and not yet released it.
$P_{1.4}$ A maximum of N locks can be acquired at any point in time, where N is a monitor parameter.
$P_{1.5}$ An acquired lock should never later be freed as memory.

The monitor is shown in Figure 1, and is defined as the class `M1` extending the `Monitor` class. The monitor is parameterized with the maximal number of locks, `limit` (line 2) that can be acquired at any point in time. A variable, `count` (line 4), is introduced to count the number of active acquisitions. The

[1] A data class is a class decorated with `@dataclass`, which allows to perform pattern matching over objects of the class, including their parameters.

body of `M1` defines a transition function (lines 7-20), and two states: `DoRelease` (lines 22-33), parameterized with a thread id and a lock id; and `DoNotFree` (lines 35-42), parameterized with a lock id. The PyContract `@data` decorator implies `@dataclass` and introduces a hash-function to store states in hash sets. These two states themselves each contain a transition function. A transition function takes as argument an event and returns a list of states (or None if not applicable).

The outermost transition function (lines 7-20) is *always* enabled, and can be perceived as representing the temporal logic box operator □. This outermost transition function, when applied to an event, will match it against two patterns: `Acquire(thread, lock)` and `Release(thread, lock)`, where `thread` and `lock` will be bound to the actual values of the incoming event. The transition corresponding to an acquisition (lines 9-17) is conditioned on the non-existence of a `DoRelease`-state with the same lock (lines 10-11), representing the fact that the lock has already been acquired by some thread (it does not matter which). If the number of active monitors is less than the limit, `count` is bumped up and a list of two states (lines 14-15) is returned, each of which is then added to the memory of the monitor. The transition corresponding to a release (lines 18-20) returns an error if no `DoRelease(thread, lock)` exists with the same thread and lock (line 19). Note how we can use a fact as a Boolean expression if all arguments are known, in contrast to the more verbose call of the `exists` predicate in lines 10-11.

An event that does not match any of the **`case`** entries is considered to not match any transition. How this is treated depends on the kind of state. Since PyContract's default is to use skip-states (in contrast to next-states), in this case it means that the event is skipped and the state stays active in memory.

The `DoRelease` state is a `HotState`, meaning that at the end of a run, an error is reported if such a state is active. In case of an acquisition, a match occurs if the second lock argument is the same as `self.lock` (the underscore '`_`' pattern matches any value). In general, any dotted name in a pattern indicates that the incoming value has to match this exact value. In case of a match, two states are returned (line 30): an error state, and the `self` state, keeping it active in the monitor. In case of a release by the thread that holds the lock, the counter is decreased and the state `ok` is returned, corresponding to removing the `DoRelease` state. The `DoNotFree` state is a normal `State` (line 36), effectively a skip state that forever monitors that the lock is not freed with a `Free` event.

2.2 Example 2

Consider the property P_2 consisting of just the first three sub-properties $P_{1.1}$, $P_{1.2}$, and $P_{1.3}$ of P_1. A consequence of this property is that acquisitions and releases of a lock must strictly alternate. The monitor `M2` in Figure 2 monitors this property. In monitor `M1`, Figure 1, line 19, we used a memory query to check that a lock is only released by the thread that acquired it. In monitor `M2` we instead use indexing (slicing) and next-states, which results in a more efficient and more succinct monitor. We shall slice on locks, meaning that for each lock encountered in the trace, PyContract will, in a hash map, map it to a monitor

```
class M1(Monitor):
    def __init__(self, limit: int):
        super().__init__()
        self.count: int = 0
        self.limit = limit

    def transition(self, event):
        match event:
            case Acquire(thread, lock)
            if not self.exists(lambda s:
              isinstance(s, M1.DoRelease) and s.lock==lock):
                if self.monitor.count < self.limit:
                    self.monitor.count += 1
                    return [M1.DoRelease(thread, lock),
                            M1.DoNotFree(lock)]
                else:
                    return error(f'limit reached')
            case Release(thread, lock)
            if not M1.DoRelease(thread, lock):
                return error(f'releasing un-acquired lock')

    @data
    class DoRelease(HotState):
        thread: str
        lock: int

        def transition(self, event):
            match event:
                case Acquire(_, self.lock):
                    return [error('already acquired'), self]
                case Release(self.thread, self.lock):
                    self.monitor.count -= 1
                    return ok

    @data
    class DoNotFree(State):
        lock: int

        def transition(self, event):
            match event:
                case Free(self.lock):
                    return error(f'Lock freed')
```

Fig. 1. The monitor M1

memory (set of states) for only that lock. All events concerning that lock are sent to only the states in that lock-specific memory. This yields two advantages: first, we do not need to search all states when an event arrives, we can just look

up and search the states for the relevant lock, and second, since only events concerning that lock are sent to that lock-specific memory, we can write our monitor more succinctly using next-states. The `Monitor` class defines a method:

```
def key(self, event) -> Optional[object]:
    return None
```

which is called on each event to return its slicing index (`None` is the default, meaning no slicing is performed). The user can override this method to indicate which events should be sliced upon. Figure 2 (lines 2-5) shows such an overriding of this method, defining the lock to be the slicing index (key) for `Acquire` and `Release` events. The second step is to define our states as next-states (lines 8 and 15). The semantics of these is that it is an error if the next observed event (sent to that state) does not match any of its transitions. A `NextState` is an acceptance state while a `HotNextState` is not.

```
1  class M2(Monitor):
2      def key(self, event) -> Optional[object]:
3          match event:
4              case Acquire(_, lock) | Release(_, lock):
5                  return lock
6
7      @initial
8      class Idle(NextState):
9          def transition(self, event):
10             match event:
11                 case Acquire(thread, lock):
12                     return M2.DoRelease(thread, lock)
13
14     @data
15     class DoRelease(HotNextState):
16         thread: str
17         lock: int
18
19         def transition(self, event):
20             match event:
21                 case Release(self.thread, self.lock):
22                     return M2.Idle()
```

Fig. 2. The monitor M2

Slicing is used in efficient runtime verification tools such as MOP [22] and QEA [26]. In PyContract the slicing criterion is, as just shown, user defined (by overriding method `key`), which allows for a more expressive form of indexing than in e.g. MOP where all data parameters are used for indexing. QEA allows for more flexible slicing criteria. Slicing is used in MOP and QEA to express past time properties, as in this example (a release must be preceded by an acquisition).

3 Comparison with Other Frameworks

Using `M2` from Figure 2, we compared PyContract against three other tools: the research tools Daut [16] and QEA [26], and the industrial tool CommaSuite [8, 21], inspired by the RV tool RuleR [21]. At [13], the implementations of `M2` in those other tools are available. We encourage the reader to compare these to the PyContract code from Figure 2.

We compared the performance of PyContract monitoring with the other tools to investigate whether it is fast enough for practical use. We ran three experiments with the four tools, using the monitors for `M2`. The first experiment simulates one thread acquiring and then immediately releasing one lock, many times. In the second experiment, one thread acquires many locks at once before releasing them. The third experiment is the same as the second except that each lock is acquired and released by a different thread. For each experiment, we recorded the processing times for traces with 500,000; 1,000,000; 2,000,000; and 4,000,000 events.

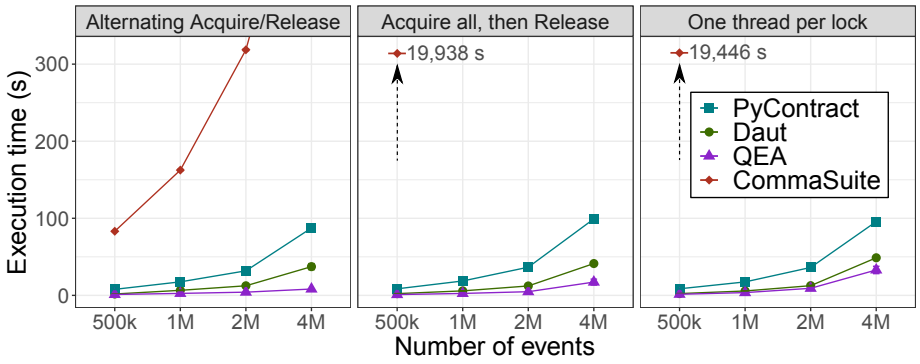

Fig. 3. Benchmark results comparing PyContract with three other tools.

Figure 3 shows the results running the experiment on an Intel Xeon E5-2680 frequency-locked to 2.40GHz. We ran PyContract 1.0.1 on Python 3.10.4 and QEA 1.0, Daut 0.1, and CommaSuite 1.1.0 on the Oracle JVM 11.0.12 running on 64-bit Linux 4.15. The graph shows the number of events in the trace (x-axis) against the time the tool took to complete offline analysis. Each point represents the mean of at least six runs. PyContract, Daut, and CommaSuite read events from JSONL files while QEA read them from a special kind of CSV file that this tool supports. The experiments and raw results are available at [13].

In the experiment, it is clear that QEA and Daut are the fastest tools, but PyContract remains usable with processing times under 100 seconds for four million events. Additionally, PyContract has similar results for all three experiments while QEA and CommaSuite are noticeably slower when many locks are acquired at once. For example, QEA with 4M events completes in about 8s when only one

lock is acquired at once but takes about 33s when 2M locks are acquired at once by 2M separate threads. One possible explanation of the superior performance of Daut over PyContract, in spite of their similar implementation, may be the superiority of the JVM and its just-in-time compilation compared to Python's runtime system.

In the cases where no CommaSuite results are reported it is because the tests took more than 320 seconds. The CommaSuite developers pointed out several potential reasons for the observed performance. Most importantly, performance has never been a main requirement[2]. CommaSuite models may be nondeterministic, which requires additional bookkeeping during monitoring. This feature cannot be disabled. Additional data is also collected to provide e.g. coverage information after monitoring; again this is not optional.

4 Conclusion

We have presented a shallow internal DSL (library) for trace analysis in Python, and argued that such DSLs have important advantages. These advantages include ease of infusion into projects due to a flattened learning curve, taking advantage of development tools (such as IDEs) and libraries for the programming language, expressiveness, and fast development and reduced maintenance making it easier to adapt to feature requests. Since Python is one of the most popular programming languages, we believe that a Python library for monitoring is valuable. We compared the performance of PyContract to two research tools and one industrial tool for RV, all three JVM-based. PyContract performed reasonably well compared to the two research tools, considering that the JVM is a high performance platform compared to Python's runtime system, and PyContract convincingly outperformed the industrial tool. The longer term objective of the library is to support activities that normally are associated with monitoring, providing a "Swiss pocket knife" for monitoring. This includes, as already mentioned, various forms of visualization, but also trace mining. Since PyContract is embedded in Python, allowing a mix of monitoring DSL and free style Python code, the door is open to experiment with the connection between runtime verification and data analysis. This line of work has already been pursued on the application to the Europa Clipper project, and we intend to pursue it further.

Acknowledgments Part of the research was funded by the ERC Advanced Grant LASSO, the Villum Investigator Grant S4OS, and DIREC, Digital Research Center Denmark. Part of the research was performed at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Part of the research was carried out as part of the Accelerando program under the responsibility of ESI (TNO) with Philips as the carrying industrial partner. The Accelerando research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

[2] CommaSuite has been designed to be used to monitor the logs from automated tests in nightly builds, where "the traces are short enough, and the nights long enough".

References

1. D. Ancona, L. Franceschini, A. Ferrando, and V. Mascardi. RML: Theory and practice of a domain specific language for runtime verification. *Science of Computer Programming*, 205:102610, 05 2021.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
3. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. 17th Int. Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 57–72, Limerick, Ireland, 2011. Springer.
4. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.
5. D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
6. K.-T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*, pages 86–91, 1993.
7. C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
8. CommaSuite. `https://projects.eclipse.org/projects/technology.comma`.
9. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of TIME 2005: the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE, 2005.
10. Daut. `https://github.com/havelund/daut`.
11. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. *Software Tools for Technology Transfer (STTT)*, 18(2):205–225, 2016.
12. Europa Clipper mission. `https://europa.nasa.gov`.
13. Experiments Repository. `https://bitbucket.org/seanmk/rv-bench`.
14. J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 339–347, New York, NY, USA, 2014. Association for Computing Machinery.
15. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
16. K. Havelund. Data automata in Scala. In *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*, pages 1–9. IEEE Computer Society, 2014.
17. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
18. S. Kauffman, K. Havelund, and R. Joshi. nfer - a notation and system for inferring event stream abstractions. In *Runtime Verification - 6th Int. Conference, RV'16*, volume 10012 of *LNCS*, pages 235–250. Springer, 2016.
19. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proc. of the 1st Int. Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

20. E. Kurklu and K. Havelund. A flight rule checker for the LADEE Lunar spacecraft. In *17th International Colloquium on Theoretical Aspects of Computing (ICTAC'20)*, volume TBD of *LNCS*, 2020.
21. I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman. Integrating interface modeling and analysis in an industrial setting. In *Proceedings of the 5th Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD'17)*, pages 345–352. SciTePress, February 2017.
22. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011. http://dx.doi.org/10.1007/s10009-011-0198-6.
23. PyContract. `https://github.com/pyrv/pycontract`.
24. PyPi Stats Website. `https://pypistats.org`.
25. Python pattern matching. `https://peps.python.org/pep-0636`.
26. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 596–610. Springer, 2015.
27. M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.
28. StackOverflow Developer Survey. `https://insights.stackoverflow.com/survey/2021`, 2021.
29. OMG Systems Modeling Language (SysML). `http://www.omgsysml.org`.
30. E. R. Tufte, N. H. Goeler, and R. Benson. *Envisioning information*, volume 126. Graphics press Cheshire, CT, 1990.
31. OMG Unified Modeling Language (UML). `http://www.omg.org/spec/UML`.