

Embedding Runtime Verification in Python

Klaus Havelund^{1*}, Moran Omer², and Doron Peled²

¹ Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA

² Bar Ilan University, Ramat Gan, Israel

Abstract. Runtime verification consists of verifying that an execution trace satisfies a specification. The specification can be formal, but it can also be informal, e.g. code in a general purpose programming language. In this paper we present the tool PYDEJAVU, which explores the boundary between general purpose programming and formal specification. The tool supports a two-phase approach to writing properties, where a property can be expressed in a combination of an operational phase and a declarative phase. The operational phase is expressed in an internal Python DSL (Domain-Specific Language), whereas the declarative phase is expressed in the external DSL QTL (Quantified Temporal Logic) for first-order past time temporal logic. This approach benefits from the expressiveness of Python and the succinctness and efficiency of monitoring QTL. The tool builds on the previous runtime verification tool DEJAVU monitoring QTL properties, and is compared with the previous TP-DEJAVU tool, which supports a purely external DSL for two-phase runtime verification.

1 Introduction

Runtime Verification (RV) allows monitoring the execution of a system, usually in the form of a trace of events, and checking it against a formal specification. The provided verdict can be used to avert a problematic behaviour, or just report the violation. Two key parameters in applying RV are expressiveness of the formalism that is used for writing specifications, and the efficiency of the monitors. Classical formalisms that are used for RV and model checking techniques are temporal logics and various kind of automata.

RV techniques (as opposed to model checking, which is a more comprehensive formal method, hence poses some further complications) have been expanded to allow temporal specification of traces of events that carry data, using first-order predicate logic constructs, including universal and existential quantification over the data values. In order to allow the online runtime verification to cope with the speed of the intercepted events, a carefully constructed compact representation

* The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the second and third authors was partially funded by Israeli Science Foundation grant 2454/23: “Validating and controlling software and hardware systems assisted by machine learning”.
© 2024. All rights reserved.

is required to keep a summary of the trace observed so far, which is updated with each new intercepted event. This entails a tradeoff between expressiveness, succinctness, and efficiency, which is typical for formal methods.

One tradeoff in selecting the specification formalism for RV is the use of a “programming language” style formalism versus a “logic-based” formalism. The use of a programming language as formalism allows a high degree of flexibility in describing the desired property. It permits using programming tricks to implement the runtime verification checks. On the other hand, a formal logic specification is usually very succinct, and can benefit from a standard efficient implementation, rather than using an ad-hoc implementation.

In previous work, we showed a method and tool for splitting the RV specification into two parts for achieving optimized expressiveness, succinctness, and efficiency. A *declarative part* is used for expressing a purely (first-order) temporal specification, which is devoid of arithmetic operations. An *operational part* is used to process intercepted events from the system under observation, and modify them, based on some local summary, before passing them to the declarative part. The operational part can embed the arithmetic operations, e.g., addition, comparison and calculating extrema values. In the tool TP-DEJAVU [15], the operational part was implemented based on a syntax that allows intercepting events and performing computations on the data, sending modified events to the RV tool DEJAVU [17]. The tool DEJAVU provides the declarative part based on a first-order past time temporal logic.

In this work, we describe a new tool, PYDEJAVU, that further enhances the capabilities of RV on traces of events with data. The contribution is to implement the operational part as an *internal DSL* (Domain-Specific Language) that is embedded in Python. Then, the declarative part is embedded in Python as an *external DSL* based on the DEJAVU system (which is implemented in SCALA). The tool can also be seen as a Python interface to DEJAVU. This allows a much higher flexibility than the small DSL for the operational part in TP-DEJAVU, and immediately opens new capabilities that were not originally allowed in TP-DEJAVU, e.g., adding further arithmetic operations, the use of all builtin and programmed functions, vectors and using more complex data structures when processing the observed events. The contribution of the tool is as follows:

- The combination of a programmable internal DSL, working in tandem with a powerful and efficient external DSL provides a large step up in the expressiveness of the RV specification. This includes vectors, functions (internal to the programming language, and user defined), the use of various data structures and even the integration of other components that are likewise embedded to work with that programming language such as Neural Networks (PyTorch [10]) and Databases (SQL [9]).
- We benefit from the succinctness and power of the external DSL part as an efficient monitor for runtime verification of events with data, based on first-order past time temporal logic.

- We demonstrate the new tool with examples and experiments. The tool is available for download from GitHub¹.

The paper is organized as follows. Section 2 provides an overview of various flavours of DSLs, and shows where the different DSLs mentioned in this paper fit in. Section 3 defines the QTL past-time temporal logic, which is the foundation for the tools presented in the paper. Section 4 gives an overview of the previous tools developed, namely DEJAVU and TP-DEJAVU. Section 5 introduces the tool PYDEJAVU, which is the key contribution of this paper. Section 6 presents experiments. Finally Section 7 concludes the paper. The paper contains an appendix A for the reviewers.

Related work

Some early tools supported data comparison and computations as part of the logic [2]. The version of the tool MONPOLY in [3] supports comparisons and aggregate operations such as sum and maximum/minimum within a first-order LTL formalism. It uses a database-oriented implementation. Other tools supporting automata based limited first-order capabilities include [4, 21]. In [7], a framework that lifts the monitor synthesis for a propositional temporal logic to a temporal logic over a first-order theory, using an SMT solver, is described. A number of pure internal DSLs (libraries in a programming language) for RV have been developed [1, 13, 14, 5, 11], which offer the full power of the host programming language for writing monitors and therefore allow for arbitrary comparisons and computations on data to be performed. Of special interest in this context is the internal Python DSL PYCONTRACT [5], which is inspired by the internal SCALA DSL DAUT [13]. The concept of phasing monitoring such that one phase produces output to another phase has been explored in other frameworks, including early frameworks such as [19] with a fixed number of phases (two), but with propositional monitoring, and later frameworks with arbitrary user defined phases [2, 12]. In particular, stream processing systems support this idea [6, 18]. A more remotely related work on increasing the expressive power of temporal logic is the extension of DEJAVU with rules described in [16].

2 Classification of Domain-Specific Languages

In this section we provide an overview of the various types of DSLs, highlighting their advantages and disadvantages. We can reveal that PYDEJAVU is a ‘*hybrid closed DSL*’ in this classification. By *host language* we mean some general-purpose programming language, such as Python, Java, Scala, C, C++, etc. We distinguish between several kinds of DSLs, as shown in Figure 1.

External DSLs are standalone languages that require their own parsers and interpreters or compilers, such as DEJAVU and TP-DEJAVU. **Internal DSLs** use the constructs of the host language and are essentially libraries in the host

¹ <https://github.com/moraneus/pydejavu>

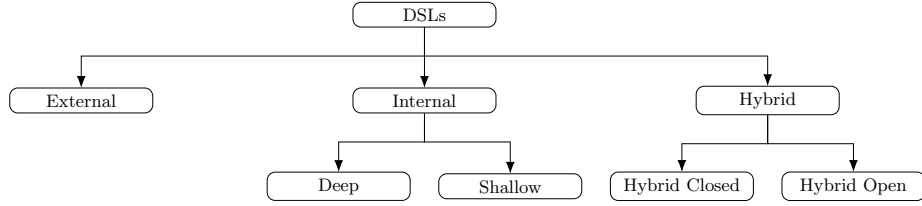


Fig. 1: Diagram of DSLs categorization

language. They come in two varieties: **Deep DSLs**, which build an Abstract Syntax Tree (AST) of the specification or program directly in the host language; and **Shallow DSLs**, which leverage the full expressiveness of the host language, making them as powerful as the host language itself. Lastly, **Hybrid DSLs** combine features of both external and internal DSLs, integrating elements from each to create a versatile language. They include **Hybrid Closed DSLs**, such as PYDEJAVU, which are primarily internal with embedded external elements that cannot refer back to the host language (an example is MySQL for Python where SQL queries are written as text strings in an external DSL), and **Hybrid Open DSLs**, where the primary language is external, and embedded internal elements are written in the host language (an example is yacc where the grammar is written in an external DSL but semantic actions are written in C).

External DSLs provide a more succinct notation for users, and they are easier to analyze, optimize, and transform. They, however, require their own parsers and machinery for interpreting them, which increases complexity, and they are usually limited in expressiveness, requiring non-trivial effort to extend expressiveness. *Internal DSLs* may generally be easier to adopt since they are “just another library” in the host language, assuming that the host language is popular. They do not require their own parsers and machinery for interpreting them, which decreases complexity. *Internal deep DSLs*, as external DSLs, are easier to analyze, optimize, and transform. However, they provide a less succinct notation for users, and they are limited in expressiveness. In the case of shallow DSLs, these are fully expressive (Turing complete). They are usually also easier to implement, and in particular modify as user requirements change, as they are directly executable, no need for implementing an interpreter or translator. However, internal shallow DSLs provide a less succinct syntax, and are much! harder to analyze, optimize, and transform, requiring meta-programming, including dealing with the AST of the host language.

3 QTL Syntax and Semantics

To motivate the development of our PYDEJAVU tool, we first introduce QTL (Quantified Temporal Logic), the formalism underlying both DEJAVU and TP-DEJAVU. This logic allows the specification of trace properties involving data and, by limiting its use to past-time operators, is suitable for interpretation over finite traces.

3.1 QTL Syntax

The formulas of the QTL logic are defined using the following grammar, where p stands for a *predicate* symbol, a is a *constant* and x is a *variable*. For simplicity of the presentation, we define here the QTL logic with unary predicates, but this is not due to a principal restriction, and in fact DEJAVU supports predicates over multiple arguments, including zero arguments, corresponding to propositions.

$$\varphi ::= \text{true} \mid p(a) \mid p(x) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \mathcal{S} \varphi) \mid \ominus \varphi \mid \exists x \varphi$$

A formula can be interpreted over multiple types (domains), e.g., natural numbers or strings. Accordingly, each variable, constant and parameter of a predicate is defined over a specific type. Type matching is enforced, e.g., between $p(a)$ and $p(x)$, where the types of the parameter of p and of a must be the same. We denote the type of a variable x by $\text{type}(x)$. *Propositional* past time linear temporal logic is obtained by restricting the predicates to be parameter-less, essentially Boolean propositions. In this case no variables, constants and quantification are needed either.

QTL subformulas have the following informal meaning: $p(a)$ is true if the last event in the trace is $p(a)$. The formula $p(x)$, for some variable x , holds if x is bound to a constant a such that $p(a)$ is the last event in the trace. The formula $(\varphi \mathcal{S} \psi)$, which reads as φ *since* ψ , means that ψ occurred in the past (including now) and since then (beyond that state) φ has been true. (The *since* operator is the past dual of the future time *until* modality in the commonly used future time temporal logic.) The property $\ominus \varphi$ means that φ is true in the trace that is obtained from the current one by omitting the last event. The formula $\exists x \varphi$ is *true* if there exists a value a such that φ is true with x bound to a . We can also define the following additional derived operators: $\text{false} = \neg \text{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\Diamond \varphi = (\text{true} \mathcal{S} \varphi)$ (“previously”), $\Box \varphi = \neg \Diamond \neg \varphi$ (“always in the past” or “historically”), and $\forall x \varphi = \neg \exists x \neg \varphi$.

3.2 QTL Formal semantics

A QTL formula is interpreted over a *trace*, which is a finite sequence of *events*. Each event consists of a predicate symbol and parameters (no variables), e.g., $p(a)$, $q(7)$. Formally, let $\text{sub}(\varphi)$, be the set of subformulas of φ . Let $\text{free}(\eta)$ be the set of free (i.e., unquantified) variables of η . The bookkeeping of which variables are mapped to what values is recorded in *assignments*, which map variables to values. Let ϵ be the empty assignment. Let γ be an assignment to the variables $\text{free}(\eta)$. We write $[v \mapsto a]$ to denote the assignment that consists of a single variable v mapped to value a . We denote by $\gamma[v \mapsto a]$ the assignment that differs from γ only by associating the value a to v . Let σ be a trace of events of length $|\sigma|$ and i a natural number, where $i \leq |\sigma|$. Then $(\gamma, \sigma, i) \models \eta$ denotes that η holds for the prefix of length i of σ with the assignment γ . We denote by $\gamma|_{\text{free}(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ . The formal semantics of QTL is defined as follows, where $(\gamma, \sigma, i) \models \varphi$ is defined when γ is an assignment over $\text{free}(\varphi)$, and $i \geq 1$.

- $(\epsilon, \sigma, i) \models \text{true}$.
- $(\epsilon, \sigma, i) \models p(a)$ if $\sigma[i] = p(a)$.
- $([x \mapsto a], \sigma, i) \models p(x)$ if $\sigma[i] = p(a)$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{\text{free}(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{\text{free}(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi \mathcal{S} \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{\text{free}(\psi)}, \sigma, j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{\text{free}(\varphi)}, \sigma, k) \models \varphi$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x \varphi$ if there exists $a \in \text{type}(x)$ such that $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

4 Previous QTL Tools

In the following section, we discuss existing tools including DEJAVU, implemented in SCALA, and its extension TP-DEJAVU, which enhances functionality with a two-phase verification process. Building upon these, we present in Section 5 the tool PYDEJAVU, which addresses limitations of DEJAVU and TP-DEJAVU.

4.1 DEJAVU

DEJAVU [17] is a runtime verification tool designed to monitor systems against QTL properties. By incrementally analyzing events, DEJAVU detects violations of temporal properties in real time. It utilizes Binary Decision Diagrams (BDDs) to efficiently represent and manipulate sets of assignments, allowing for compact storage and fast evaluation of logical formulas. DEJAVU supports both offline trace analysis and live monitoring, handling data-driven specifications by mapping variable bindings to Boolean values through BDDs.

The DEJAVU tool uses keyword characters to express QTL formulas; it employs the following notation: **forall** and **exists** stand for \forall and \exists , respectively. **P**, **H**, **S**, and **@** correspond to \Diamond , \Box , \mathcal{S} , and \ominus , respectively. Additionally, **|**, **&**, and **!** represent \vee , \wedge , and \neg , respectively.

Example 1. We will illustrate DEJAVU with an example. This example will be used later to demonstrate the tools TP-DEJAVU and PYDEJAVU. Consider a simple filesystem mechanism that handles several key events. The filesystem allows opening a file with the an **open(F, f, m, s)**, carrying as arguments the folder name **F**, the filename **f**, the access mode **m** (read or write), and the size **s** which is the maximal number of bytes that can be written. The **close(f)** event indicates that a file *f* has been closed. The write event **write(f, d)** contains the filename and the data **d** (a string) being written. We shall assume that new data are appended to the existing file, which is not important for this example but it will be for a subsequent example in PYDEJAVU. Additionally, the system supports **create(F)** and **delete(F)** events, which represent the creation or deletion of a folder.

The requirement we are verifying states that if data is written to a file, the file must have been opened in write mode, not closed since, and must reside in

a folder that has been created and not deleted since. The formalization of this property is shown below in a mathematical QTL-like format. In Figure 2 the property is shown using DEJAVU syntax.

$$\forall f \forall d \text{ write}(f, d) \rightarrow \left(\begin{array}{l} \exists F \exists s \\ \left((\neg \text{close}(f) \mathcal{S} \text{ open}(F, f, "w", s)) \right) \\ \wedge (\neg \text{delete}(F) \mathcal{S} \text{ create}(F)) \end{array} \right)$$

```
forall f . forall d .
  write(f, d) ->
    (exists F . Exists s .
      ((! close(f) S open(F, f, "w", s)) & (! delete(F) S create(F))))
```

DEJAVU Syntax

Fig. 2: Example 1 - DEJAVU

Suppose we now wanted to express a property about the number of data bytes written over time. Expressing such dynamic file size conditions cannot be accomplished with DEJAVU, as it requires arithmetic operations, such as calculating totals (sum), which DEJAVU does not support. TP-DEJAVU was created for this purpose.

4.2 TP-DEJAVU

The DEJAVU tool provides a powerful and efficient RV algorithm for monitoring events with data, based on the QTL logic. However, it has some deficiencies in supporting specifications that include e.g., arithmetic operations. It does allow using basic arithmetic comparisons, but their use tames down the strong optimization of the DEJAVU monitoring, which is based on enumerating data values, representing them as bit-vectors and using BDD operations on sets of bit-vectors. The reason is that comparisons are performed between the original values, not the bit-vectors representation of enumeration. This deficiency is related to the particular DEJAVU representation, which focuses on logical operators rather than arithmetic operators.

This gave the motivation for a “two phase” RV monitoring, as implemented in TP-DEJAVU [15, 22]. The first phase is *operational*. It performs *preprocessing* of the monitored events. The specification of the operational part is given as set of small procedures in a programming language like style syntax. The procedures process a *summary* of the trace of events seen so far, based on the type and values of the data of the most recent intercepted (observed) event. Then the event can be modified, e.g., augmented with newly calculated values, before forwarding it to the second phase. The operational phase is in particular effective for comparing

between the most recent values, and calculating aggregation, e.g., summing up observed data values or calculating the maximal value.

The second phase is *declarative*, as it expresses the property in terms of a temporal specification over the global view of the trace seen so far rather than in terms of how the recent event changes the summary of events seen so far². This phase is supported by the DEJAVU tool. In TP-DEJAVU, the two phases, the operational and declarative, are implemented together as one monolithic tool, programmed in the programming language SCALA. In fact, the operational part is written as an extension of the DEJAVU tool. An additional feature allows the operational part to use the intermediate verdicts calculated by the declarative part on the trace intercepted so far in preprocessing the next event. This makes the interaction between the two phases bi-directional.

This tool is controlled with a completely rigid external DSL, hence it is naturally susceptible to a fixed set of predefined features that can be used, including operators, functions, data structures and programming constructs. If a need to extend the capabilities of the tool occurs, e.g., adding the use of collections (vectors, maps, sets) of data that appear in observed events, some non-trivial implementation effort would be required.

Example 2. As in Example 1, our goal is to ensure that all requirements for writing data into a file are met. Additionally, we can leverage the operational phase to verify that the total number of bytes written to all files together does not exceed a specified max value. To achieve this, we need to implement a counting mechanism. The TP-DEJAVU specification is shown below in Figure 3.

The TP-DEJAVU DSL supports the operational and declarative phases. It includes *initiate* to define the initial state, including variables, their types and their initial values. This is followed by *on* clauses that specify actions to be executed when various events are observed. Within each *on* clause, variables can be updated, and the *output* keyword is used to generate modified events for further processing by DEJAVU in the second phase. The syntax allows for arithmetic operations, Boolean logic, and data manipulations, enabling TP-DEJAVU to preprocess events and potentially modify them for temporal runtime verification checks performed by DEJAVU.

In this specification the variable `total` keeps track of the total number of bytes written. Upon observing an open event, the open event is resubmitted to DEJAVU without the size argument since it is not needed. Upon observing a write event, the total is augmented and a write event is returned to DEJAVU with a Boolean second argument being true only if the total number of bytes is still below the limit. The DEJAVU formula requires that this value must never be FALSE.

² In fact, the declarative part, as in other tools based on temporal logic, hides a transformation to an update of the summary. However, this update is complex and powerful, and cannot be used by itself as a specification formalism that replaces the declarative part.

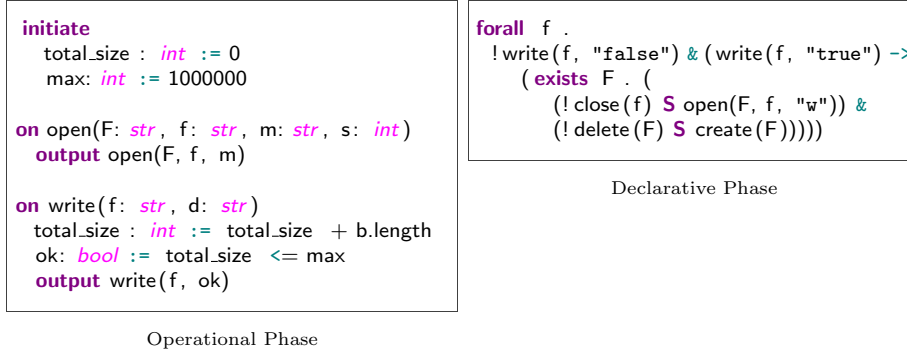


Fig. 3: Example 2 - TP-DEJAVU

5 PYDEJAVU

PYDEJAVU [20] is a Python library (≥ 3.8) providing a bridge between Python and the SCALA-based DEJAVU [8] runtime verification tool. It is designed for two-phase Runtime Verification (RV) processing, combining the flexibility and expressiveness of Python with the rigorous and efficient, declarative monitoring capabilities of the DEJAVU tool. Simplified stated, PYDEJAVU replaces the operational phase DSL in TP-DEJAVU with Python procedures; these procedures are embedded within a fixed interface that wraps them, providing the interface between the operational part and the declarative part, see Figure 4. Although DEJAVU is implemented in SCALA, we chose Python as the front end language due to its widespread use³, thereby potentially reaching a larger audience for runtime verification.

The communication between the Python component and DEJAVU is facilitated by the third-party library *pyjnius*. Pyjnius is a Pythonic JAVA Native Interface (JNI) that enables JAVA code running in a JAVA Virtual Machine (JVM) to invoke and be invoked by Python applications. Adopting this approach necessitated some changes to DEJAVU, including configuration of DEJAVU from within Python, result feedback to Python, and creation of a result file.

PYDEJAVU embodies the integration of an external DSL (DEJAVU’s QTL) with an internal DSL, with Python hosting the internal DSL. By using Python, users can perform any desirable operations on events, including arithmetic, string manipulations, and Boolean logic. Additionally, Python’s extensive data structures and objects can be leveraged for data storage and complex calculations. This flexibility allows PYDEJAVU to manage a wide range of operational tasks during runtime, giving users the ability to customize event processing and analysis to suit their specific needs.

Example 3. This example demonstrates how PYDEJAVU provides additional expressiveness, compared to DEJAVU (Example 1) and TP-DEJAVU (Example 2).

³ Python is by several sources evaluated to be the most popular programming language at the time of writing, see e.g. <https://spectrum.ieee.org/top-programming-languages-2024>.

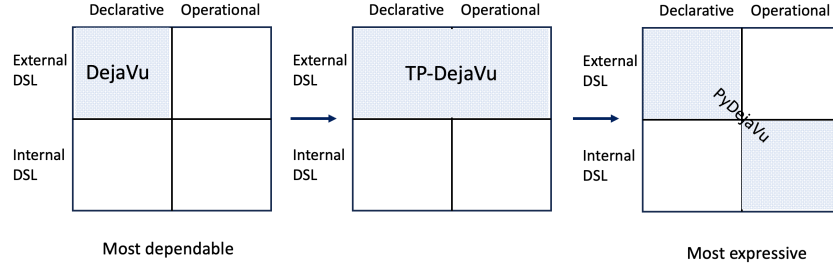


Fig. 4: The evolution of QTL tools.

We augment the original property in Example 1 with the property that there is a limit on how many bytes can be written to each file. For this we need to store the current number of bytes stored in each file. This is done in the specification below by maintaining a global variable `total_size`, which is a Python dictionary, mapping file names to their current sizes. This cannot be expressed in TP-DEJAVU without extending that system, or taking care of it outside the system.

The monitor is initialized as an instance of the `Monitor` class, part of the PY-DEJAVU framework, which must be imported. This monitor takes a specification written in QTL as a text string as input. Serving as the core engine, the monitor tracks events issued from the operational part in Python during program execution and ensures they align with the QTL specification. The event handlers, such as `open`, `close`, and `write`, are decorated with the `@event` annotation, which binds them to specific events in the runtime. That is, if a function `f` is annotated with `@event("e")` then that function will be called on all events with name `e`. Each such annotated event handler function returns a list, where the first item is the event name and subsequent items represent values associated with that event. The returned list is then submitted to DEJAVU as an event. For example, in the `open` event, the function returns `["open", Folder, file, mode]`, where `Folder`, `file` and `mode` are the arguments passed to the handler, representing the folder where the file located, the file being opened and the mode in which it is opened. Similarly, the `write` function returns `["write", ok]`, where `ok` is a Boolean flag indicating whether the file size exceeds a defined limit.

Additionally, if an event handler returns `None`, it means that nothing is forwarded to the declarative part, and the system waits for new events. Event handlers are not required for every event, e.g., for `create` and `delete`. In these cases, events submitted to the monitor are directly passed to DEJAVU, ensuring that the default processing behavior is maintained without additional modifications.

6 Experiments

We present experimental results on the relative efficiency of runtime monitoring QTL specifications using the TP-DEJAVU tool compared to the combined internal and external RV tool PYDEJAVU. In all cases, both specifications pro-

```

from pydejavu.core.monitor import Monitor, event

specification = """
prop example: forall f .
  !write(f, "false") & (write(f, "true") ->
    (exists F . ((!close(f) S open(F, f, "w")) & (!delete(F) S create(F))))
"""

monitor = Monitor( specification )
total_sizes : dict[ str, int ] = {}

@event("open")
def open(F: str, f: str, m: str, s: int):
    global total_sizes
    if mode == "w":
        total_sizes[f] = s
    return ["open", F, f, m]

@event("close")
def close(f: str):
    global total_sizes
    del total_sizes[f]
    return ["close", f]

@event("write")
def write(f: str, d: str):
    global total_sizes
    if f not in total_sizes :
        total_sizes[f] = 0
    data_len = len(d)
    ok = total_sizes[f] >= data_len
    if ok:
        total_sizes[f] -= data_len
    return ["write", f, ok]

events = [
    {"name": "create", "args": ["tmp"]},
    {"name": "open", "args": ["tmp", "f1", "w", "10"]},
    {"name": "write", "args": ["f1", "some text"]}
]

for e in events:
    monitor.verify(e)
monitor.end()

```

Fig. 5: Example 3 - PYDEJAVU

duced identical outputs. Our benchmarks focused solely on time and memory consumption during the evaluation phase, excluding compilation time.

Properties The QTL properties used in our experiments are as follows in Figure 6:

1. **forall** x . (p(x) -> **exists** y . (**P** q(x, y) & y > 10))
2. **forall** x . **forall** y . ((p(x) & @q(y) & x < y) -> **P** r(x, y))
3. **forall** x . (p(x) -> (**forall** y . (@P q(y) -> x > y) & **exists** z . @ **P** q(z)))

Fig. 6: Experimental Properties

Their adaptations for TP-DEJAVU and PYDEJAVU are shown in Appendix A (Figures 7 and 8, respectively).

Property	Method	Trace 10K	Trace 100K	Trace 500K	Trace 1M	Trace 5M
$P1$	TP-DEJAVU	0.39s 105MB	0.91s 311MB	2.91s 646MB	5.45s 1.34GB	44.86s 4.01GB
	PYDEJAVU	0.50 174MB	1.21 435MB	4.55s 1.24GB	8.85s 2.26GB	63.90s 3.88GB
$P2$	TP-DEJAVU	0.39s 110MB	0.98s 297MB	3.02s 707MB	5.48s 1.06GB	59.14s 4.23GB
	PYDEJAVU	0.62s 155MB	1.38s 431MB	4.75s 957MB	9.26s 1.37GB	84.14s 3.43GB
$P3$	TP-DEJAVU	0.41s 143MB	0.95s 311MB	2.52s 672MB	4.74s 960MB	31.68s 3.91GB
	PYDEJAVU	0.53s 172MB	1.47s 434MB	4.89s 1.00GB	9.33s 1.69GB	59.16s 3.67GB

Table 1: DEJAVU vs. TP-DEJAVU vs. PYDEJAVU.

Traces The event sequences and their corresponding values were randomly generated using Python scripts to ensure a diverse range of scenarios for all properties. In our experiment, we employed traces containing varying events to test our approach’s scalability and performance. Specifically, we utilized traces with 10K, 100K, 500K, 1M, and 5M events. This range allowed us to observe the behavior of the system under different levels of complexity and load.

Results Table 1 presents the results from the experiments, indicating that TP-DEJAVU is slightly faster than PYDEJAVU. Notably, DEJAVU was unable to evaluate any of the traces except one, as its execution times consistently exceeded 1000 seconds. These results are somewhat predictable, given that Python is likely less efficient than SCALA. We actually anticipated that TP-DEJAVU would be even faster relative to PYDEJAVU: while TP-DEJAVU executes its operational phase and declarative phase in the same language (SCALA), PYDEJAVU involves two languages, Python and SCALA. This dependency introduces additional overhead due to the communication between the operational phase in Python and the declarative phase in SCALA.

7 Conclusion

The true strength of PYDEJAVU, which combines both internal and external DSLs, lies in its remarkable flexibility for executing complex calculations, even in scenarios where the specifications may seem challenging or impractical for TP-DEJAVU. This flexibility stems largely from Python’s clear, familiar, and expressive syntax, enabling developers to easily construct intricate logic. Additionally, the embedded declarative component, based on DEJAVU, leverages PY-DEJAVU’s capabilities to perform complex relational calculations, particularly for handling the first-order past-time aspects of the desired specifications. A further advantage of embedding RV in Python is the possibility of integrating other libraries such as e.g. SQL and PyTorch. Future work can include implementing in SCALA interface to DEJAVU. Another line of work includes investigating a hybrid open DSL, where a QTL formula can directly refer to functions in the host programming language.

References

1. Howard Barringer and Klaus 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.
2. Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for runtime monitoring: From eagle to ruler. In Oleg Sokolsky and Serdar Taşiran, editors, *Runtime Verification*, pages 111–125, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
3. David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.*, 46(3):262–285, 2015.
4. Christian Colombo, Andrew Gauci, and Gordon J. Pace. Larvastat: Monitoring of statistical properties. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 480–484. Springer, 2010.
5. Dennis Dams, Klaus Havelund, and Sean Kauffman. A Python library for trace analysis. In Thao Dang and Volker Stolz, editors, *22nd International Conference on Runtime Verification (RV)*, volume 13498 of *LNCS*, page 264273. Springer International Publishing, 2022.
6. B. D’Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME’05)*, pages 166–174, 2005.
7. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transf.*, 18(2):205–225, 2016.
8. DEJAVU tool source code. <https://github.com/havelund/dejavu>.
9. Andy Dustman. Python MySQL. <https://pypi.org/project/MySQL-python/>, 2024.
10. Github. Pytorch. <https://github.com/pytorch/pytorch>, 2024.
11. Felipe Gorostiaga and César Sánchez. Hstriver: A very functional extensible tool for the runtime verification of real-time event streams. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 563–580. Springer, 2021.
12. Sylvain Halle and Roger Villemare. Runtime enforcement of web service message contracts with data. volume 5, pages 192–206, 2012.
13. Klaus 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.
14. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
15. Klaus Havelund, Panagiotis Katsaros, Moran Omer, Doron Peled, and Anastasios Temperekidis. Tp-dejavu: Combining operational and declarative runtime verification. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 249–263, Cham, 2024. Springer Nature Switzerland.

16. Klaus Havelund and Doron Peled. An extension of first-order LTL with rules with application to runtime verification. *Int. J. Softw. Tools Technol. Transf.*, 23(4):547–563, 2021.
17. Klaus Havelund, Doron Peled, and Dogan Ulus. First-order temporal logic monitoring with bdds. *Formal Methods in System Design*, 56(1-3):1–21, 2020.
18. Hannes Kallwies, Martin Leucker, Malte Schmitz, Albert Schulz, Daniel Thoma, and Alexander Weiss. Tessla – an ecosystem for runtime verification. In Thao Dang and Volker Stolz, editors, *Runtime Verification*, pages 314–324, Cham, 2022. Springer International Publishing.
19. Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. In Klaus Havelund and Grigore Rosu, editors, *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001, Paris, France, July 23, 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 218–235. Elsevier, 2001.
20. PYDEJAVU tool source code. <https://github.com/moraneus/pydejavu>.
21. Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. Marq: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.
22. TP-DEJAVU tool source code. <https://doi.org/10.5281/zenodo.8322559>.

A Specifications in Experiment

This appendix is for reviewers and will not be included in the final paper. It will be made available on the PYDEJAVU GitHub page [20].

<pre> on q(x: <i>int</i>, y: <i>int</i>) in_bound: <i>bool</i> := y > 10 output ite(in_bound, q(x), skip) forall x . (p(x) -> P q(x)) </pre>	<pre> initiate last_q: <i>bool</i> := false on p(x: <i>int</i>) x.lt.y: <i>bool</i> := last_seen_q && x < y last_q: <i>bool</i> := false output p.q(x, y, x.lt.y) on q(y: <i>int</i>) last_q: <i>bool</i> := true output skip on r(x: <i>int</i>, y: <i>int</i>) last_q: <i>bool</i> := false output r(x,y) forall x . forall y . (p.q(x, y, "true") -> P r(x, y)) </pre>	<pre> initiate max_y: <i>int</i> := -1 on p(x: <i>int</i>) x.gt.y: <i>bool</i> := x > max_y output p(x, x.gt.y) on q(y: <i>int</i>) is_max: <i>bool</i> := max_y < y max_y: <i>int</i> := ite(is_max, y, max_y) output q(y) forall x . forall w . (p(x, w) -> (p(x, "true") & exists y . @ P q(y))) </pre>
Property 1	Property 2	Property 3

Fig. 7: TP-DEJAVU properties: operational (above) and declarative (below).

<pre> @event("q") def handle_q(arg_x: int, arg_y: int): if arg_y > 10: return "q", arg_x return None forall x . (p(x) -> P q(x)) </pre>	<pre> last_seen_q = False y = 0 @event("p") def handle_p(arg_x: int): nonlocal y, last_seen_q x_lt_y = last_seen_q and arg_x < y last_seen_q = False return "p.q", arg_x, y, x_lt_y @event("q") def handle_q(arg_y: int): nonlocal y, last_seen_q y = arg_y last_seen_q = True @event("r") def handle_r(arg_x: int, arg_y: int): nonlocal last_seen_q last_seen_q = False return "r", arg_x, arg_y forall x . forall y . (p.q(x, y, "true") -> P r(x, y)) </pre>
Property 1	Property 2
<pre> max_y = -1 @event("p") def handle_p(arg_x: int): nonlocal max_y x_gt_y = arg_x > max_y return "p", arg_x, x_gt_y @event("q") def handle_q(arg_y: int): nonlocal max_y max_y = max(max_y, arg_y) return "q", arg_y forall x . forall w . (p(x, w) -> (p(x, "true") & exists y . @ P q(y))) </pre>	
Property 3	

Fig. 8: PYDEJAVU properties matching the TP-DEJAVU properties in Figure 7.