Embedding Monitoring of First-order Temporal Logic in a Programming Language

Klaus Havelund[1]*, Moran Omer[2], and Doron Peled[2]

[1] Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, USA
[2] Bar Ilan University, Ramat Gan, Israel

Abstract. Runtime verification (RV) consists of verifying that an execution trace satisfies a specification. The specification can be written in a formal logic, or it can be written as code in a general purpose programming language. In this paper we present an RV method, and a corresponding tool, that explores the boundary between formal logic and general purpose programming. The tool, called PyDejaVu, 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. Our tool builds on the previous runtime verification tool DejaVu monitoring Qtl properties.

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. Three key parameters in applying RV are expressiveness of the formalism that is used for writing specifications, the succinctness of the formalism, 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 techniques used in model checking, which is a more comprehensive formal method, hence posses 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

* 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".

existential quantification over the data values. Online RV monitors a sequence of events that are emitted by the checked system as it executes. In order to allow the online RV to cope with the speed of the intercepted events, a carefully constructed compact representation 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 [15], we showed a method and tool, called TP-DejaVu, 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 operations on data, including e.g. 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 arithmetic operations, e.g., addition, comparison and calculating extrema values. In the tool TP-DejaVu, 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 [18]. 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, and generally using builtin and programmed functions, as well as using more complex data structures, such as vectors, 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 functions (builtin and user defined), the use of various data structures, and even the integration of other Python libraries 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, optimized 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 GitHub[1].

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.

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, 25]. 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 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 [23] 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, 21]. Another related work on increasing the expressive power of temporal logic is the extension of DEJAVU with rules described in [17].

2 Classification of Domain-Specific Languages

In this section, we provide an overview of the various types of Domain-Specific Languages (DSLs), highlighting their advantages and disadvantages. DSLs are specialized languages designed to simplify the expression of domain-specific tasks by providing constructs that closely align with particular domain concepts. This focus allows DSLs to enhance productivity and reduce errors compared to general-purpose programming languages.

[1] https://github.com/moraneus/pydejavu

DSLs play a significant role in runtime verification by offering tailored constructs that make it easier to specify and monitor system behaviors. They can be categorized into three main types based on their interaction with a host language. A host language refers to a general-purpose programming language, such as Python, Java, or Scala, that provides the underlying environment in which a DSL operates. The three types of DSLs are: external DSLs, which are completely separate from the host language they are implemented in (i.e., no host-language code is written by a user); internal DSLs, which are embedded directly within the host language (the specification language is the host language); and hybrid DSLs, which combine elements of both, offering a balance between flexibility and integration. Figure 1 provides a visual overview of this classification.

Before we provide a more detailed explanation of these concepts, we can reveal that PyDejaVu falls under the category of *hybrid closed DSLs*. It leverages the strengths of both external and internal approaches to provide a structured yet adaptable way to express domain-specific rules.

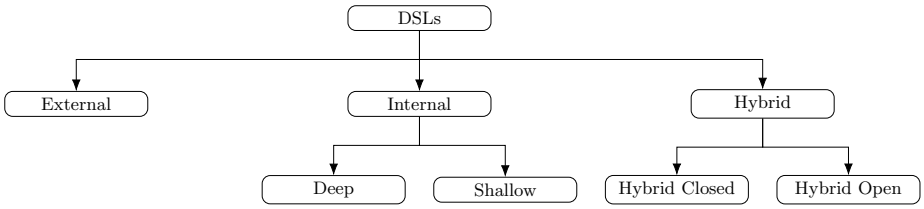


Fig. 1: DSL categorization.

2.1 External DSLs

External DSLs are stand alone languages with dedicated grammars, parsers, and interpreters or compilers, designed to address specific domain requirements independently of any host language. Systems like DejaVu and TP-DejaVu exemplify this approach by providing specialized syntax for runtime verification. For example, TP-DejaVu implements its operational phase using a distinct grammar, as shown in Figure 4, offering a domain-specific abstraction separate from general-purpose programming languages.

These languages provide users with a concise, domain-focused syntax that simplifies analysis, optimization, and transformations. They enhance reliability by making specifications more formal and interpretable. However, external DSLs come with increased implementation complexity due to the need for dedicated parsing and interpretation machinery. Additionally, their rigid structure makes it challenging to adapt to evolving requirements, a phenomenon known as "requirement creep", where new or unforeseen needs necessitate extending the languages capabilities. As discussed in [16], this can e.g. involve adding support for complex operations like arithmetic comparisons or timing constraints, features without which the language may become too limited for practical use.

2.2 Internal DSLs

Internal DSLs represent a powerful approach to domain-specific language implementation that leverages the existing infrastructure of a host programming language. Unlike external DSLs that require dedicated parsers and interpreters, internal DSLs are implemented as libraries within the host language, making them more accessible and maintainable. These DSLs come in two distinct varieties.

Deep internal DSLs represent specifications or programs as data structures within the host language. This design choice means that a formula by the user is specified as an object of a data type. An external DSL is usually parsed from text into an abstract syntax tree, which is then processed further. In a deep internal DSL, we skip the parsing of text, and the specification writer manually creates the abstract syntax tree, potentially using various auxiliary functions. The implementation in [14] demonstrates this approach, presenting a runtime verification DSL that is mostly deep in nature but extends the paradigm by allowing code as part of the specification. Such deep DSLs share most of the qualities of external DSLs in that they are easier to analyze, optimize, and transform, though they provide less succinct notation (than external DSLs) for users and are limited in expressiveness (as external DSLs).

Shallow internal DSLs, in contrast, embrace the full expressiveness of the host programming language. The DAUT Scala library [19] is an example of a shallow internal DSL. Written in Scala, it takes advantage of Scala's powerful pattern-matching capabilities and object-oriented features. Consider the example in Figure 2. The monitor class CommandMonitor inherits from the class Monitor[Event], demonstrating how the internal DSL seamlessly integrates with the host language's type system and inheritance mechanisms. The implementation leverages Scala's pattern matching for event processing and condition checking. This implementation has been successfully used in practice, including in mission-critical applications[2].

Deep DSLs offer better analytical properties and optimization potential but sacrifice expressiveness. Shallow DSLs provide full computational power but make static analysis and optimization more challenging, often requiring sophisticated meta-programming techniques to reason about the code. Their integration into existing development environments and tool chains makes them particularly attractive for scenarios where seamless interoperability with existing code is prioritized over domain-specific syntactic optimization.

2.3 Hybrid DSL

Hybrid DSLs represent a language design approach that combines features of both external and internal DSLs to leverage their respective strengths while

[2] The monitor verifies that whenever (always) a command dispatch is observed with a task id, a command number, and a time, then within 20 time units a completion must be observed, with no other dispatch of the same task id and command number observed in between.

```
class CommandMonitor extends Monitor[Event] {
  always {
    case Dispatch(taskId, cmdNum, time1) =>
      hot {
        case Dispatch(`taskId`, `cmdNum`, _) => error
        case Complete(`taskId`, `cmdNum`, time2)
          if time2 - time1 <= 20 => ok
      }
  }
}
```

Fig. 2: Example - shallow DSL.

mitigating their limitations. We categorize them into two distinct types: Hybrid Closed DSLs and Hybrid Open DSLs.

Hybrid Closed DSLs are external DSLs that are used/called from within a programming language. An illustrative example is Python's MySQL library [9], which allows a Python program to execute MySQL statements provided as text strings. This is a hybrid solution since the DSL is invoked from a programming language, and closed since from within the DSL there is no reference back to the programming language. One may refer to such DSLs as *programming language first* DSLs. A PyDejaVu example is shown in Figure 6.

Hybrid Open DSLs go in the opposite direction by allowing an external DSL to contain statements in a programming language, typically within special brackets. One may refer to such DSLs as *programming language second* DSLs. The UNIX `yacc` parser generator [20] exemplifies this approach, where the grammar is specified in an external DSL while semantic actions are implemented in C witin curley brackets. This is a hybrid solution since the programming language is invoked from the DSL, and open since from within the DSL there is reference back to the programming language. The aspect-oriented AspectJ language [22] is another example of a hybrid open DSL, since aspects (external DSL) can contain Java code.

3 QTL Syntax and Semantics

To motivate the development of our PyDejaVu tool, we first introduce QTL (Quantified Temporal Logic), the formalism underlying 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.

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

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 $\mathit{type}(x)$. *Propositional* past time linear temporal logic is obtained by restricting the predicates to be parameterless, 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: $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\diamondminus\, \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ ("previously"), $\boxminus\, \varphi = \neg \diamondminus\, \neg\varphi$ ("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 $\mathit{sub}(\varphi)$, be the set of subformulas of φ. Let $\mathit{free}(\varphi)$ be the set of free (i.e., unquantified) variables of φ. 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 $\mathit{free}(\varphi)$. 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 $(\gamma, \sigma, i) \models \varphi$ denotes that φ holds for the prefix of length i of σ with the assignment γ. We denote by $\gamma|_{\mathit{free}(\varphi)}$ 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 $\mathit{free}(\varphi)$, and $i \geq 1$.

- $(\epsilon, \sigma, i) \models \mathit{true}$.
- $(\epsilon, \sigma, i) \models p(a)$ 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 $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi \,\mathcal{S}\, \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{\mathit{free}(\psi)}, \sigma, j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{\mathit{free}(\varphi)}, \sigma, k) \models \varphi$.

– $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
– $(\gamma, \sigma, i) \models \exists x\ \varphi$ if there exists $a \in \mathit{type}(x)$ 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 [8, 18] 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 $\diamondminus$, $\boxminus$, $\mathcal{S}$, 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 event open(F, f, m, s), carrying as arguments the folder name F, filename f, access mode m (read or write), and maximum writable size s in bytes. 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. 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 3 the property is shown using DejaVu syntax.

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

```
prop example1:
  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. 3: 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, as mentioned, 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-vector representation of enumerations. 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 [26]. The first phase is *operational*. It performs *preprocessing* of the monitored events. The specification of the operational part is given as a collection of small procedures in a programming language like syntax. The procedures maintain a *summary* of the trace of events seen so far, based on the argument values of the intercepted (observed) events. An 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 values, and for performing calculations, such as aggregation, e.g., summing up observed data values or calculating the maximal value.

The *operational* phase of a TP-DejaVu specification includes an *initiate* section defining 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.

The *declarative* phase expresses the properties in terms of temporal specifications over the sequence of modified events forwarded by the operational part. This phase is supported by the DejaVu tool. In TP-DejaVu, the two phases, operational and declarative, are implemented together as one monolithic tool, programmed in the programming language Scala. 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.

The supported formalism is a purely 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 4.

```
initiate
  total_size: int := 0
  max: int := 1000000

on open(F: str, f: str, m: str, s: int)
  output open(F, f, m)

on write(f: str, d: str)
  total_size: int := total_size + b.length
  ok: bool := total_size <= max
  output write(f, ok)
```

Operational Phase

```
prop example2:
  forall f .
    !write(f, "false") &
    (write(f, "true") ->
      (exists F . (
        (!close(f) S open(F, f, "w")) &
        (!delete(F) S create(F)))))
```

Declarative Phase

Fig. 4: Example 2 - TP-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.

5 PyDejaVu

PyDejaVu [24] is a Python3 library providing a bridge between Python and the Scala-based DejaVu [8, 18] runtime verification tool. It is, as TP-DejaVu, designed for two-phase processing, but combining the flexibility and expressiveness of Python with the rigorous and efficient, declarative monitoring capabilities of the DejaVu tool. Simply stated, PyDejaVu replaces the operational phase DSL in TP-DejaVu with Python. Specially decorated functions provide the interface between the operational part and the declarative part. Although DejaVu is implemented in Scala, we chose Python as the front end language due to its widespread use[3], 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 Python library that uses the Java Native Interface (JNI) to enable 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. Figure 5 shows the evolution of the three tools DejaVu, TP-DejaVu, and PyDejaVu presented in this paper, all based on the Qtl logic.

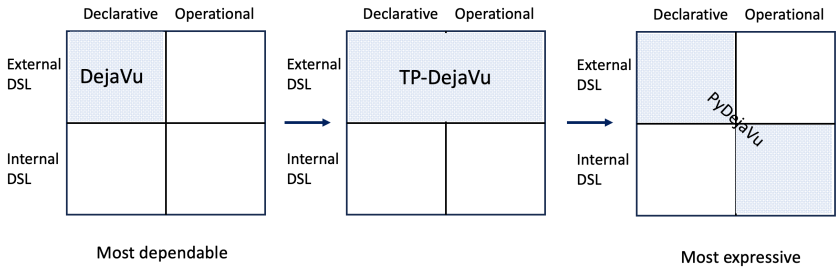


Fig. 5: The evolution of Qtl tools.

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.

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 available space for each file. This is done in the specification below by maintaining a global variable available_space, which is a Python dictionary, mapping file names to their available space. This cannot be expressed in TP-DejaVu without extending that system, or taking care of it outside the system.

```
from pydejavu.core.monitor import Monitor, event

specification = """
  prop example3:
      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)
available_space: dict[str, int] = {}

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

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

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

# ----------------------
# Applying monitor to an example trace:
# ----------------------

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. 6: Example 3 - PyDejaVu.

The monitor in Figure 6 is initialized as an instance of the Monitor class, part of the PyDejaVu library, which must be imported. This monitor takes as argument a string representing a specification in Qtl. 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 handler functions, such as open, close, and write, are decorated with the @event annotation[4], which binds them to specific events. That is, if a function 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", F, f, m], where F, f and m 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", f, ok], where f is the file being written to, and ok is a Boolean flag being true iff. the resulting file size is within the size limit provided when the file was opened.

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. The experiments were performed on an Apple MacBook Pro laptop with an M1 Core processor, 16GB RAM, and 512GB SSD storage, running the macOS Sonoma operating system. In all cases, both specifications produced identical outputs. Our benchmarks focused solely on time and memory consumption during the evaluation phase, excluding compilation time.

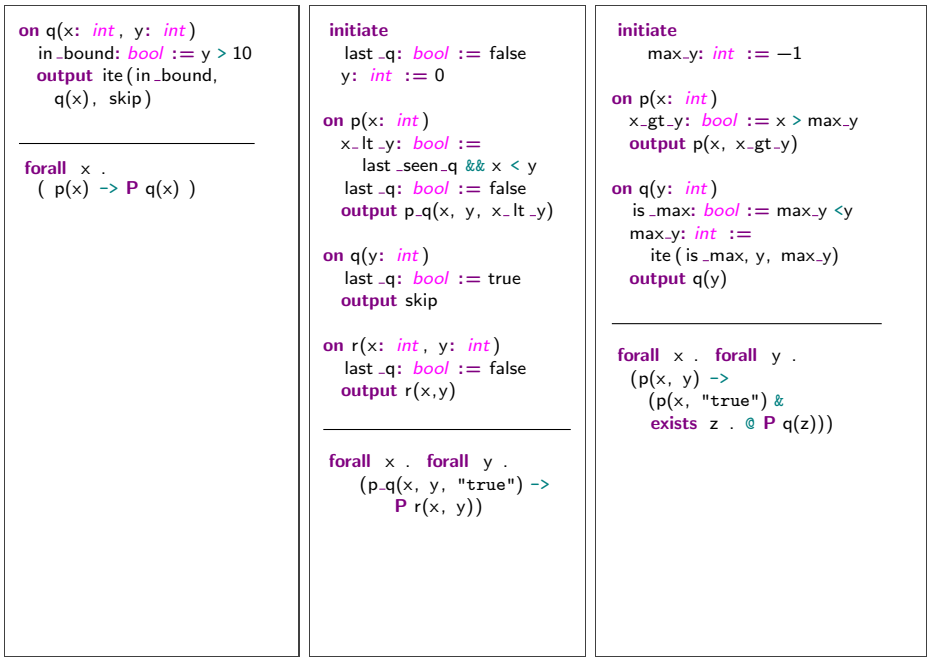
Properties The Qtl properties used in our experiments are shown in Figure 7. Their adaptations for TP-DejaVu and PyDejaVu are shown in Figures 8 and 9, respectively.

Traces In our experiment, we utilized traces containing varying events to test our approach's scalability and performance. The event sequences and their corresponding values were generated using Python scripts that employed a mix of

[4] In Python, one can define a function D that takes a function as an argument and returns a new function, we call D a decorator. If a function g is decorated with D using the @-sign (i.e., @D), then g is effectively replaced by $D(g)$. In the case of @event("open"), the call event("open") returns a decorator that modifies the function defined below it.

```
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. 7: Experimental Properties.



Property 1 Property 2 Property 3

Fig. 8: TP-DejaVu properties: operational (above) and declarative (below).

random methods and minimal modeling, with constraints to ensure variety, control over trace characteristics, and a diverse range of scenarios for all properties. Specifically, we used 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.

```
@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) )
```

Property 1

```
    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))
```

Property 2

```
    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 y .
  (p(x, y) ->
    (p(x, "true") &
    exists z . @ P q(z)))
```

Property 3

Fig. 9: PyDejaVu properties matching the TP-DejaVu properties in Figure 8.

Property	Method	Trace $10K$	Trace $100K$	Trace $500K$	Trace $1M$	Trace $5M$
$P1$	DejaVu	11.35s 1.21GB	∞ -	∞ -	∞ -	∞ -
	TP-DejaVu	0.39s 105MB	0.91s 311MB	2.91s 646MB	5.45s 1.34GB	44.86s 4.01GB
	PyDejaVu	0.50s 174MB	1.21s 435MB	4.55s 1.24GB	8.85s 2.26GB	63.90s 3.88GB
$P2$	DejaVu	∞ -	∞ -	∞ -	∞ -	∞ -
	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
$P3$	DejaVu	∞ -	∞ -	∞ -	∞ -	∞ -
	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: Time and memory usage.

7 Conclusion

The true strength of PyDejaVu, which combines both internal and external DSLs, lies in its flexibility for executing complex calculations, especially in scenarios where the specifications are challenging or impossible to express in 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 leverages 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 a 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 Taşıran, 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 Villemaire. 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, Moran Omer, and Doron Peled. Operational and declarative runtime verification (keynote). In *Proceedings of the 7th ACM International Workshop on Verification and Monitoring at Runtime Execution*, VORTEX 2024, page 312, New York, NY, USA, 2024. Association for Computing Machinery.
17. 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.
18. 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.
19. Klaus Haveund. Daut - Monitoring Data Streams with Data Automata. https://github.com/havelund/daut, 2024.
20. Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. https://en.wikipedia.org/wiki/Yacc.
21. 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.

22. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 327–354. Springer, 2001.
23. 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.
24. PyDejaVu tool source code. https://github.com/moraneus/pydejavu.
25. 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.
26. TPDejaVu tool source code. https://doi.org/10.5281/zenodo.8322559.