DSLs for Runtime Verification

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) allows monitoring executions of systems against formal specifications. A major challenge in increasing the capabilities and scope of formal methods stems from the tradeoff in increasing the expressiveness of the specification formalism used, while taming down the complexity of the involved algorithms and preserving the succinctness of the specifications. The focus of RV on a single execution at a time allows great flexibility in the way RV is implemented and towards achieving these goals. We focus here on the possibilities for implementing RV logics as external DSLs (Domain-Specific Languages), internal DSLs, and hybrid DSLs - a mix of the two. We also address the use of AI to generate monitors from natural language requirements. We survey the possibilities and focus in particular on the effect it has on achieving a desired level of expressiveness. A concrete challenge on which we focus here is allowing the use of arithmetic operations and relations on data that appear in the monitored events.

1 Introduction

Runtime Verification (RV) includes the monitoring of system executions, checking them against formal specifications. Three key parameters in applying RV are Expressiveness of the formalism that is used for writing specifications, Elegance (including succinctness) of the formalism, and the Efficiency of the verification process, which we refer to as *the three* E*'s*. Among formal methods, RV is one of the most open-ended in that it allows for writing monitors in a spectrum of languages, ranging from dedicated domain-specific languages (logics) to general-purpose programming languages. It involves instrumenting the code to emit a sequence of events that can be analyzed and compared against a (potentially formal) specification. Focusing on a single trace provides flexibility in choosing the specification formalism. The selection of the formalism needs to accord with the possibility of maintaining efficiency, in particular when RV is applied online and needs to keep track with the pace of the intercepted events. There is also an advantage in employing "standard" specification formalisms, used broadly

* 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".

and interpreted uniformly and being able to express the specification property succinctly. There is often a tradeoff between the three E's. A classical example of such a tradeoff is between using propositional LTL and monadic first order logic [37]; both have the same expressiveness, only that the complexity of deciding the latter is nonelementary higher than the former, but can allow specifications that are nonelementary shorter than the former.

It is not surprising then to witness that there is a large spectrum of RV tools. We survey here alternatives for implementing RV logics as Domain-Specific Languages (DSLs), namely *external*, *internal* and *hybrid* DSLs. In brief, an external DSL is a, usually "tiny", language with its own grammar and parser, existing independently from the programming language it is implemented in. In contrast, an internal DSL is embedded in a programming language, e.g., as a library. A hybrid DSL is a mix of the two. One tradeoff in selecting the specification formalism for RV is between the use of a "programming language" style formalism versus a "logic-based" formalism. The use of a programming language as a 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 succinct, and can benefit from an efficient implementation. It also makes it easier to convince oneself that the specification conforms to what was intended.

We provide examples of RV tools that fit the corresponding DSL categories, and outline advantages and disadvantages. We specifically compare the fitness of different approaches for implementing RV tools w.r.t. the ability to support specifications that include the following ingredients:

- Reasoning about traces with events that contain data.
- Comparing between data items that appear in different events with respect to first-order quantification (*exists*, *forall*).
- Using functions and relations from signatures such as the natural numbers, reals, strings, etc. in the specification.

In general, obtaining the three ingredients (E's) together, in their full optima, might not easily be an achievable goal. One can observe a progress from using propositional specification, as in, e.g., [25] into first-order specification [24, 5] and then adding a restricted ability of using objects interpreted over integers or strings [21]. As a forward-looking perspective, the emergence of Large Language Models (LLMs) may bring us closer to achieving the three E's by allowing the generation of efficient monitors from expressive abstract high level natural language specifications. This will also be discussed.

The paper is organized as follows. Section 2 provides some comments on moving from propositional to first-order specification in the fields of formal verification and runtime verification. Section 3 provides an overview of the various flavors of DSLs. Section 4 describes a file system and its requirements, which will be used as a running example to illustrate the different approaches. Section 5 discusses external DSLs. Section 6 discusses internal DSLs. Section 7 discusses hybrid DSLs. Section 8 discusses the relevance and application of LLMs to RV synthesis. Finally, Section 9 concludes the paper.

Related work

Several RV tools have been developed over time, and giving a complete overview would be a daunting task. In this section, we only mention a few which are not otherwise discussed in this paper. Some early tools supported data comparison and computations as part of the logic, including, e.g. RuleR [4] (external DSL). The version of the tool MonPoly (external DSL) in [5] 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 that support limited first-order capabilities based on automata include MarQ [36] (external DSL) and LarvaStat [7] (hybrid DSL). In [10] a framework is described that elevates the monitor synthesis for a propositional temporal logic to a temporal logic Tdl (external DSL) over a first-order theory, using an SMT solver, and implemented in the Junit$^{\mathrm{RV}}$ tool. Several internal DSLs for RV have been developed that 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. These include TraceContract [3], Daut [18], LogFire [19], and BeepBeep [17]. Stream processing systems such as Lola [9] and TeSSLa [28] (external DSLs), and HStriver [16] (hybrid DSL) offer expressive DSLs, and support the concept of communicating monitors. Another related work on increasing the expressive power of temporal logic is the extension of DejaVu with rules (external DSL) described in [23].

2 From Propositional to First-Order Specification

In this section we relate RV to the field of formal verification w.r.t. the expressiveness of the specification. In both fields we can observe a transition from propositional specifications to first-order specifications over data. In formal verification this quickly leads to undecidability, whereas in RV the concern is usually related to efficiency.

Verifying the correctness of systems is, in general, undecidable. Even the basic problem of proving that a given sequential algorithm terminates on a particular input is undecidable. In the early 80s it was observed that limiting the focus of verification to finite state systems, with a careful selection of a specification formalism, in particular based on propositional temporal logics or finite automata, allows applying effective decision procedures [14, 34], which are now referred to as *model checking*. Restricting the verification to a finite model is a nontrivial limitation; it prohibits verifying algorithms that are based on data structures such as trees or queues, where the size of the data structure is not restricted. This also applies to algorithms that operate on unrestricted numerical values or strings. Nevertheless, model checking has gained a huge success and it is often sufficient to use a finite state abstraction, or to, at least, find errors by checking an implementation of the software with concrete restrictions on the size of the memory or the number of bits that is used to store numerical values.

One way to extend model checking beyond the Boolean based temporal logic and state representation is based on Bounded Model Checking (BMC). For finite state spaces and propositional based specification, BMC can use a SAT solver. Extending the scope beyond finite state systems calls for using SMT (Satisfiability Modulo Theory) solvers [2]. Still, inherent undecidability of the satisfiability for common domains (based on Gödel incompleteness for the naturals), forces limitations on this approach: either by using decidable domains, or by allowing the occasional failures of the SMT solvers (e.g., in the sense that the operation may time out).

In runtime verification, one concentrates on a single execution at a time. This is a more modest target than model checking, which addresses the entire set of executions of a system. This raises the hopes for additional capabilities of RV, especially w.r.t. expressiveness and efficiency. Indeed, progressing from propositional based specification and events encoded as Boolean minterms into first-order specification and events with data is achievable with quite efficient tools and algorithms. For example, MonPoly [5] and DejaVu [24] are based on first-order (past) linear temporal logic and allows events with data[1].

Adding the ability to use functions and relations from a signature that includes e.g., the natural numbers or strings is another leap in expressiveness that needs to be dealt with carefully. This capability can entail undecidability in RV. For example, it is undecidable, in general, to calculate a verdict for Diophantine equations [30], e.g., written in the form $\forall y\,(p(y) \rightarrow \exists x\,\exists z\,(x^2 + z^2 + y = 0))$, where $p(y)$ refers to an event with some value set to the variable y ($p(y)$ is used to make the decision depend on the input trace). In fact, this is undecidable with a single event of the form $p(y)$. (Although one can sometimes write a monitor for a *particular* equation, in particular for the above example.) However, such undecidable specifications are very unlikely to be the focus of RV in practice. Instead, going from propositional to first-order specification in RV can make the calculation of the verdict complex to compute. Consider for example: $\forall x\ (p(x) \rightarrow \exists y \exists z (\diamondminus\ q(y) \wedge\ \diamondminus\ q(z) \wedge y \neq z \wedge x = (y+z)/2))$, where $\diamondminus$ means *previously in the past*. Verifying such a property is hard to implement efficiently, since the property requires that a new p event has a value that is the average of two distinct values observed in previous q events. This means remembering and comparing the new value of all p events against all the previous values that appeared in previously observed q events, which is growing unboundedly. This defies optimizations that are typical to RV tools with a rigid syntax (implemented as an external DSL). But it can be expressed within a tool that is open for "programming" the way we would express this property in an internal DSL or a hybrid DSL.

3 Classification of Domain-Specific Languages

In this section, we provide an overview of the various types of DSLs, highlighting their advantages and disadvantages. DSLs are specialized languages designed to simplify the expression of domain-specific tasks by providing constructs that

[1] MonPoly supports assertions about a fixed number of future steps.

closely align with particular domain concepts. This focus allows DSLs to enhance productivity and reduce errors compared to general-purpose programming languages.

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 domain specificity and flexibility. Figure 1 provides a visual overview of this classification.

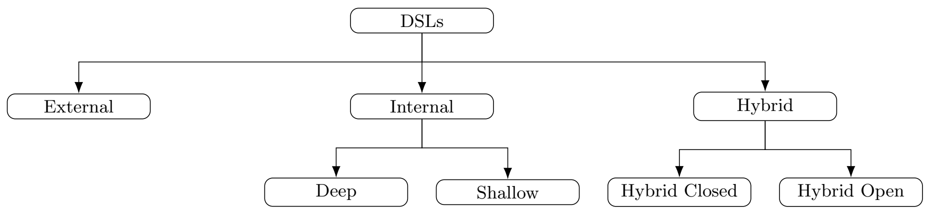

Fig. 1: DSL categorization.

3.1 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. 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 analyzable. 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 language's capabilities. As discussed in [22], 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. We illustrate some of the advantages and difficulties in using external DSLs based on the systems DejaVu [11, 24] and TP-DejaVu [21, 38].

3.2 Internal DSLs

Internal DSLs leverage 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 [19] 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. BeepBeep [17] is another example. 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). We do not present an example of a deep internal DSL in this paper.

Shallow internal DSLs, in contrast, embrace the full expressiveness of the host programming language. 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. From an implementation perspective, internal DSLs typically reduce development overhead by eliminating the need for separate parsing and interpretation infrastructure. However, this comes at the cost of syntax flexibility, the DSL must conform to the host language's syntactic constraints. This limitation often leads to less concise specifications compared to purpose-built external DSLs. The success of internal DSLs in practice can be highly dependent on the programming language chosen. E.g., an internal DSL in Python is more likely to be adopted than an internal DSL in Cobol. Their integration into existing development environments and toolchains makes them particularly attractive for scenarios where seamless interoperability with existing code is prioritized over domain-specific syntactic optimization. Internal DSLs can be well suited for more general data analysis, where the result of monitoring are data of arbitrary data types, rather than just Boolean yes/no/don't know verdicts [12]. We illustrate this category with the PyContract [8, 32] Python RV library.

3.3 Hybrid DSL

Hybrid DSLs represent a language design approach that combines features of both external and internal DSLs to leverage their respective strengths while 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 [13], 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. We illustrate this category with the PyDejaVu [22, 33] DSL.

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 [27] exemplifies this approach, where the grammar is specified in an external DSL while semantic actions are implemented in C within curly 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 [29] is another example of a hybrid open DSL, since aspects (external DSL) can contain Java code. We do not present an example of a hybrid open DSL in this paper. RV examples include LarvaStat [7] and HStriver [16].

4 The File System Example

In this section we introduce an example that will be used to demonstrate the different approaches. The example is in particular meant to illustrate the need for different levels of expressiveness of the DSLs, and to illustrate the difference w.r.t. elegance of expression. Consider a simple UNIX-like file system, where a user can create and delete folders, open and close files in folders, and write to and read from files. The concrete events that we can monitor are the following.

`create(F)`	Create a folder `F`.
`delete(F)`	Delete a folder `F`.
`open(F,f,m,s)`	Open file `f` in folder `F`, with access mode `m`, and if write mode, a maximum writable size `s` in bytes.
`close(f)`	Close file `f`.
`write(f,d)`	Write data `d` (a string) to file `f`.
`read(f)`	Read from file `f`.

Access modes are `read` or `write`. If the mode is `write`, the size parameter `s` indicates how many bytes are maximally allowed to be written to the file. In the case of `read` mode it is irrelevant.

We can now formulate the requirements that we want to monitor using the different DSLs.

R_{folder}	A file must be opened in a folder that has been created and not deleted since.
R_{write}	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.
R_{total}	The total number of bytes written to all files must not exceed a specified `max` value.
R_{size}	The number of bytes written to a file must not exceed the `size` parameter of the open event.
R_{close}	A file that is opened must eventually be closed.

The properties R_{folder} and R_{write} are past time properties. Property R_{close} is a future time property. These properties only require knowledge of identity of data observed in events. Properties R_{total} and R_{size} require storing and comparing data values. Property R_{total} only needs one summary value. Property R_{size} is slightly more complicated as it requires storing and comparison of data values per file.

5 External DSLs

In this section we present two external DSLs, namely DejaVu, a pure temporal logic, and TP-DejaVu which adds operational features expanding expressiveness.

5.1 The DejaVu Temporal Logic

The tool DejaVu [11, 24] is an external DSL. Its specification language is QTL, which is first-order past linear temporal logic. A DejaVu specification uses relation symbols to represent events. The relation symbols are uninterpreted, and the only real relation is equivalence. In fact, equivalence is intrinsic to the specification by the multiple use of a variable name within the same scope of quantification rather than explicit, e.g., by using the standard = notation.

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.

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

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: $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\Diamond\!\!\!-\ \varphi = (\mathit{true}\, \mathcal{S}\, \varphi)$ ("previously"), $\boxminus\ \varphi = \neg\ \Diamond\!\!\!-\ \neg\varphi$ ("always in the past" or "historically"), and $\forall x\ \varphi = \neg\exists x\ \neg\varphi$.

QTL **Formal semantics**. A QTL formula is interpreted over a *trace*, which is a finite sequence of *events*, with positions numbered $1, 2, \ldots$. Each event consists of a predicate symbol and parameters (no variables), e.g., $p(a)$, $q(7)$. 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 $1 \leq i \leq |\sigma|$. 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 $1 \leq i \leq |\sigma|$.

- $(\epsilon, \sigma, i) \models \mathit{true}$.
- $(\epsilon, \sigma, i) \models p(a)$ 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 $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

As a representative DEJAVU specification, consider the property R_{write} (property R_{folder} is easily expressible as well and simpler). Informally, R_{write} asserts 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 property is formalised below in a QTL-style mathematical notation and in Figure 2 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 Rwrite:
  forall f . forall d .
    write(f, d) ->
      (exists F . exists s .
        ((! close(f) S open(F, f, "w", s)) & (! delete(F) S create(F))))
```

Fig. 2: DejaVu - an external declarative DSL.

The tool DejaVu allows applying several optimizations on the implementation, which facilitates efficient monitoring. RV over propositional logic, where there is a finite number of events that can be encoded as Boolean combinations, requires only a finite amount of memory summarizing the observed trace, which is independent of the length of the inspected trace. Progressing into first-order temporal logic over events with data, such a summary cannot be bounded in length, as is obvious from the fact that in the above simple specification we must keep the growing set of values (e.g. file names, in this case) that appeared along the trace. However, the rigidity of DejaVu permits optimizations that tame the size of the summary and the incremental complexity needed to update it each time a new event appears. This is done based on the following principles:

1. The values that appear in events are mapped into short bitstrings. This can be done, e.g., by enumerating the values by natural numbers as they appear, and then map these into their binary representation (note that the original values need to be stored so that subsequent occurrences of the same value are mapped to the same bitstring).
2. Subformulas in the summary of the inspected trace correspond to relations over values, where the number of arguments equals the number of free variables in the subformula. These are encoded as BDDs over the bitstring representation.
3. Updating summaries can be performed by applying BDD operations.

These implementation steps, together with other tricks used (e.g., applying some garbage collection to the utilized bit vectors) tame down the amount of memory required to process very long traces of events, where one can easily process traces with hundreds of thousands of events. It also permits, in many cases, efficient processing that facilitates online runtime processing.

However, expanding the expressiveness of the specification language to include data type operator signatures, such as integer and string operators (e.g. $\leq$, $+$, ...), while maintaining the efficiency of the algorithm, is a challenge.

5.2 The TP-DejaVu Extension of DejaVu

A solution that goes well with an external DSL RV tool is to select a small set of constructs that extend the expressiveness of the used formalism, while maintaining the efficiency. Such an extension is implemented in the tool TP-DejaVu [21, 38]. Extending DejaVu is limited to the use of a small number of relations over the selected signatures of integers and strings. Moreover, the use of these relations, when comparing values from different events, is limited to a fixed (small) distance from the current intercepted event. This allows implementing the extension by keeping a limited amount of additional information about the previous events. Although the allowed additional relations are fixed, there is a great flexibility in combining them. In fact, this is done by writing a small procedure in a syntax that resembles a simple programming language, that is executed in between intercepting the event and calling DejaVu. This code can alter the event. In particular, it can use its finite memory and calculations to perform aggregation, e.g., calculating sums or maxima.

Specifically, TP-DejaVu consists of two components, operating in a pipeline: a new *operational* component, and DejaVu, which is referred to as a *declarative* component. The operational component has capabilities of using relations and functions from the given signatures (e.g., integers, strings, etc.). Upon interception of an event, the operational component makes some calculations that can involve applying comparisons with events from a fixed distance (e.g., previous event, two events before, etc.), which involve some storage whose size is fixed, i.e., independent of the size of the observed trace. The code depends on the relation that appears in the input event, e.g., the procedure for processing the input event `open(F, f, m, s)` is different from the procedure for processing the event `close(f)`. As a result of the calculation, a modified event is generated and sent to the declarative component. The latter just processes the new event according to the DejaVu logic and correspondingly updates the verdict. The code for writing the operational procedures is implemented in its own external DSL, extending the first-order temporal specification of the declarative component.

The following example shows how TP-DejaVu enforces the requirements R_{write} and R_{total}. As in the pure DejaVu setting, we must guarantee that every write operation satisfies the file- and folder-integrity conditions. The added operational stage lets us maintain a running counter of the bytes written across all files and flag a violation once the cumulative total exceeds a user-defined limit max. Figure 3 contains the complete TP-DejaVu specification, combining the counting logic in the operational phase with the declarative property checked by DejaVu.

In this specification, the variable total_size keeps track of the total number of bytes written so far. When an `open` event is intercepted, it is forwarded to DejaVu without its size argument, because that value is irrelevant to the safety conditions. On each `write` event, total_size is incremented by the length of the payload; the event is then resubmitted to DejaVu with a Boolean flag ok that is *true* iff total_size has not yet exceeded the threshold max. The DejaVu formula requires that this value must never be *false*.

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

Operational Phase

```
prop Rwrite_total :
   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. 3: TP-DejaVu - an external operational + declarative DSL.

Adding the operational layer to DejaVu allows a *controlled* addition of relations and functions from domains associated with rich signatures without extending the temporal logic to permit undecidability of providing a verdict. While gaining expressiveness, the elegance of the formalism, now injecting "programming" as part of the specification, is reduced. As a consequence, it is possible to write properties that less obviously represent the intension of the specification. Furthermore, the constructs that are allowed by this DSL are rather limited, and some further features are not permitted, e.g., the flexibility to use expandable vectors so that multiple maxima can be accumulated and used over the process of monitoring.

6 Internal DSLs

An internal DSL is a library in a general-purpose programming language, referred to as the host language. We distinguished in Section 3 between deep internal DSLs and shallow internal DSLs. In this section we shall write a monitor for all the properties presented in Section 4 using the shallow internal DSL PyContract [8, 32], which is a Python library for writing monitors. PyContract is inspired by the Daut [18, 20] and TraceContract [3] Scala libraries based on similar ideas. All of these are again inspired by rule-based programming as introduced in the RuleR system [4] in that the memory of a monitor is a set of facts, where a fact in its basic form is a named data record. However, unlike traditional rule systems seen e.g. in expert systems, facts in PyContract, like states in state machines, can have transitions which, upon triggering, can generate other facts, while removing the fact whose transition is taken. The important idea is that a state can be parameterized with data, which is not the case in traditional automata theory.

The library. The library provides a class `Monitor`, which any monitor must subclass. This class has the following structure from an outside usage point of view:

```
class Monitor:
  def eval(self, event: object): ... # contents omitted
  def end(self): ...                 # contents omitted
  def verify(self, trace: list[object]):
    for event in trace:
      self.eval(event)
    self.end()
  ...
```

Conceptually a monitor stores a set of active states. As we shall see (Figure 5), a user can create an instance `m` of a monitor, call `m.eval(e)` for each observed event `e`, and finally optionally call `m.end()`. Each such call of `m.eval(e)` will cause the monitor to call an `s.eval(e)` method on each active state `s`, which can either result in a new set of states, or an unchanged state. The `m.end()` method checks if there are any so-called *hot* states (see below) in the memory, and produces error messages if so. These represent bounded future time properties, things that should have happened but did not at the end of the trace in case the trace is finite. An alternative, if a whole finite trace `t` is available, is to just call `m.verify(t)`.

The events. Monitors can monitor events of any legal Python type (all types subclass type `object`). We will in this case program the monitor to process dictionaries as events, specifically dictionaries representing the six file operations:

```
{'name': 'Create', 'folder': str}
{'name': 'Delete', 'folder': str}
{'name': 'Open'  , 'folder': str, 'filename': str,
                   'mode'  : str, 'size': int}
{'name': 'Close' , 'filename': str}
{'name': 'Write' , 'filename': str, 'data': str}
{'name': 'Read' , 'filename': str}
```

As we shall see, we can perform pattern matching over such dictionaries. Another common style is to alternatively define each event type as a data class, which also allows to perform pattern matching over such event objects.

The monitor. Our monitor class FileMon in Figure 4 inherits from the class Monitor. We deliberately program the monitor as a mixture of traditional programming and temporal programming to show that the two paradigms can live together. Some monitoring problems are just easier to just program. In a traditional manner, to handle properties R_{folder} and R_{write}, we introduce two monitor local variables: `max` (Line 6), storing initially the maximal number of bytes that can be written to all files, a parameter to the monitor, and `folders` (Line 7), storing what folders are open at any time.

The monitor defines a `transition` function (Line 9), which is applied to every event that is submitted to the monitor with the `eval` method, and which can be overridden by the user as shown here. The actual type of the `transition` method is:

```
 1 from pycontract import *
 2
 3 class FileMon(Monitor):
 4   def __init__(self, max: int):
 5     super().__init__()
 6     self.max = max
 7     self.folders: set[str] = set()
 8
 9   def transition(self, event):
10       match event:
11         case {'name': 'Create', 'folder': folder}:
12           self.folders.add(folder)
13         case {'name': 'Delete', 'folder': folder}:
14           self.folders.discard(folder)
15         case {'name': 'Open', 'folder': folder, 'filename': filename, 'mode': mode, 'size': size}:
16           self.check(folder in self.folders, f'Folder {folder} not created')
17           return FileMon.File(folder, filename, mode, size)
18         case {'name': 'Write', 'filename': filename, 'data': data}:
19             if not self.exists(FileMon.File, name=filename):
20           return error(f'file {filename} not opened')
21
22   @data
23   class File(HotState):
24     folder: str
25     name: str
26     mode: str
27     size: int
28
29     def transition(self, event):
30       match event:
31         case {'name': 'Delete', 'folder': self.folder}:
32           return error('folder deleted')
33         case {'name': 'Close', 'filename': self.name}:
34           return ok
35         case {'name': 'Write', 'filename': self.name, 'data': data}:
36           self.check(self.mode == 'write', f'File {self.name} not opened in write mode')
37           self.check(len(data) <= self.size, f'File {self.name} size exceeded')
38           self.check(len(data) <= self.monitor.max, f'Total bytes written exceeded max value')
39           self.monitor.max -= len(data)
40           return FileMon.File(self.folder, self.name, self.mode, self.size - len(data))
```

Fig. 4: PyContract - a shallow Python DSL.

```
def transition(self, event: object) ->
      Optional[State | List[State]]
```

The function returns either `None` (corresponding to no match, in which case the monitor remains unchanged), a state, or a list of states. Conceptually the result of monitoring an event is a list of new states, that all become active, and all of which must lead to success corresponding to a conjunction. A single returned state s is transformed to a singular list $[s]$.

The outermost `transition` function pattern matches on the incoming event, In the case of folder creation and deletion it just updates the `folder` variable. In the case of a file opening, it checks that the folder exists, and then returns a new state (Line 17), an object of the class `File`, representing the fact that this file has been opened. This class (state) is defined as an inner class of the monitor

(Line 23). Note that if multiple files are open at the same time, multiple such states will exist. In the case of a write event, we check that there indeed exists such a `File` state for that file (Line 19), and if not an error state is returned. Note that we can, in this way, query the monitor memory for the existence of a state.

The `File` state itself (Line 23) is defined as a `HotState`, meaning that it is recorded as an error if such a state exists when the `end()` method is called. It is parameterized with the folder the file is created in, the file name, the access mode, and the maximal number of bytes that can be written to the file. The `transition` function (Line 29) returns an error if the folder is deleted while the `File` state exists. On the other hand, the state is peacefully removed (represented by `ok`) if the file is closed. Finally, in the case of a write event, it is checked that it was opened in write mode and that the total size as well as the file sizes are respected. A new `File` state is then returned (Line 40) with the remaining size allowed.

Using the monitor. Figure 5 illustrates how the monitor can be applied to analyze a trace of events. Events can be fed, one by one, using the `eval(event: object)` method. In the case of a finite sequence of observations, for example when examining a log file, a call of the `end()` method tells the monitor that the sequence has ended. Note that `end()` may not be called when monitoring is online, but if it is called, any outstanding obligations that have not been satisfied (expected events that did not occur) will be reported as errors.

```python
m = FileMon(1000000)
trace = [
  {'name': 'Create', 'folder': 'folder1'},
  {'name': 'Open', 'folder': 'folder1', 'filename': 'file1', 'mode': 'write', 'size': 1000},
  {'name': 'Write', 'filename': 'file1', 'data': 'data1'},
  {'name': 'Write', 'filename': 'file1', 'data': 'data2'},
  {'name': 'Close', 'filename': 'file1'},
  {'name': 'Delete', 'folder': 'folder1'}
]
m.verify(trace)
```

Fig. 5: Using the monitor.

Other features. PyContract offers many other features, such as always-states (always active), or-states (representing disjunction), not-states (representing negation), sequence-states (representing sequencing), next-states (failing if no transition cases match an event), grouping of monitors, and user-defined indexing (slicing) to optimize monitoring, similar to what is supported in RV systems such as MOP [31] and QEA [35]. Finally, monitors can be visualized using PlantUML.

7 Hybrid DSLs

PyDejaVu [22, 33] is a Python library that refactors TP-DejaVu's external operational DSL into standard Python while retaining the original two-phase architecture. In the operational phase, every intercepted event is routed to a user-defined function marked with the decorator `@event`. Because these handlers are written in unrestricted Python, they can exploit the full Python language ecosystem such as loops, rich data structures such as lists, dictionaries, and sets, comprehensions over such, higher-order functions, and external libraries, before emitting a transformed event. The declarative phase remains the unmodified DejaVu monitor, implemented in Scala and supporting the Qtl temporal logic [11, 24], as described in Section 5.1. Although DejaVu is implemented in Scala, Python was chosen as the front-end language due to its widespread use[2].

Communication between the two phases is mediated by PyJNIus, a JNI bridge that allows the Python runtime to instantiate the JVM-hosted DejaVu engine, push events, adjust configuration parameters on the fly, and retrieve verdicts or result files. Compared with TP-DejaVu, this design greatly enlarges the space of computable pre-processing tasks (e.g., complex aggregation, pattern matching, or vector manipulation) while preserving the proven efficiency of DejaVu's temporal reasoning core.

Figure 6 demonstrates the additional expressiveness that PyDejaVu offers over both the plain DejaVu and TP-DejaVu examples. Besides enforcing R_{write}, the specification now incorporates R_{size}, which limits the number of bytes that can be written to each file individually. This per-file accounting is realized in the Python layer by a global dictionary available_space that maps every open file to its remaining quota, information that TP-DejaVu cannot represent, and which would require an extension of the framework to capture.

The monitor is created with `monitor = Monitor(specification)`, where the argument is a Qtl specification given as a string. During execution, each incoming event triggers a handler function whose name is bound to the event by the @event annotation[3]. The handler may read or update available_space, perform arbitrary computations, and then return either a list such as ["write", f, ok], which is forwarded to the declarative DejaVu core, or None, which suppresses forwarding. For events that need no preprocessing, like `create` or `delete`, no handler is required; those events flow directly to DejaVu, preserving the default behaviour.

[2] Python is by several sources evaluated to be the most popular programming language at the time of writing [26].

[3] In Python, one can define a function D that takes a decorable object (such as a function, method, class, ...) as an argument and returns a new object, 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.

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

specification = """
 prop Rwrite_size :
   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 m == "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: PyDejaVu - a hybrid closed DSL.

8 Generating Monitors with LLMs

It is always possible to program a monitor in a general-purpose programming language without the use of a DSL (internal or external). However, such monitors may be laborious to program and it may be harder to convince oneself that they are correct. However, with the emergence of Large Language Models (LLMs) the situation may be changing. In this section we shall illustrate how a file system monitor can be generated in Python from a natural language specification, using no specific monitoring library. We used the Windsurf LLM-enhanced Visual Studio IDE [39], using the Gemini 2.5 Pro LLM [15]. Specifically, we provided a version of Section 4 as a prompt. In addition, we described the dictionary event types. Finally we asked it to use pattern matching over the events.

It responded with a long version, which was too large to fit a figure in the paper. We therefore asked it to refactor common code patterns into an abstract `Monitor` class, which can be seen in Figure 7. We also asked it to shorten error messages to fit the paper width. However, the logic of the monitor stayed the exact same during these refactorings. The `FileSystemMonitor` can be seen in Figure 8.

```python
class Monitor(abc.ABC):
    @staticmethod
    def require(condition: bool, message: str):
        if not condition:
            raise Exception(message)

    def process_events(self, events: list[dict]):
        for i, event in enumerate(events):
            try:
                self.process_event(event)
            except Exception as e:
                raise Exception(f"Event {i+1} failed: {e}") from e
        self.finalize()
        print("OK")

    @abc.abstractmethod
    def process_event(self, event: dict):
        pass

    @abc.abstractmethod
    def finalize(self):
        pass
```

Fig. 7: Abstract Monitor class generated by the Gemini 2.5 LLM.

We notice that the `FileSystemMonitor` checks for more requirements than we originally requested, all of which, however, make perfect sense (we left these out originally in Section 4 to keep the problem small). The monitored requirements in addition to those in Section 4 are (Figure 8):

– A folder cannot be created if it is already created and not deleted (Line 11)
– A folder must exists in order to be deleted (Line 15).
– A file cannot be opened if it is already open (Line 21).
– A file cannot be closed if it is not open (Line 37).
– A file cannot be read unless it is open (Line 41).

There are several ways in which LLMs can be used for generating monitors. In [1] an experiment is described where two LLM generated versions of a program are checked against each other using differential testing. In [6] an approach is

```
class FileSystemMonitor(Monitor):
    def __init__(self, max_total_bytes: int):
        self.max_total_bytes = max_total_bytes
        self.total_bytes_written = 0
        self.created_folders = set()
        self.open_files = {}

    def process_event(self, event: dict):
        match event:
            case {'name': 'Create', 'folder': folder}:
                self.require(folder not in self.created_folders, "folder exists")
                self.created_folders.add(folder)

            case {'name': 'Delete', 'folder': folder}:
                self.require(folder in self.created_folders, "folder missing")
                self.created_folders.remove(folder)

            case {'name': 'Open', 'folder': folder, 'filename': filename,
                  'mode': mode, 'size': size}:
                self.require(folder in self.created_folders, "folder missing")
                self.require(filename not in self.open_files, "file open")
                self.open_files[filename] = {'mode': mode, 'max_size': size, 'bytes_written': 0}

            case {'name': 'Write', 'filename': filename, 'data': data}:
                self.require(filename in self.open_files, "file closed")
                file_info = self.open_files[filename]
                self.require(file_info['mode'] == 'write', "write mode")
                data_size = len(data)
                self.require(file_info['bytes_written'] + data_size <= file_info['max_size'],
                             "size limit")
                self.require(self.total_bytes_written + data_size <= self.max_total_bytes,
                             "total limit")
                file_info['bytes_written'] += data_size
                self.total_bytes_written += data_size

            case {'name': 'Close', 'filename': filename}:
                self.require(filename in self.open_files, "file closed")
                del self.open_files[filename]

            case {'name': 'Read', 'filename': filename}:
                self.require(filename in self.open_files, "file closed")

            case _:
                raise ValueError("bad event")

    def finalize(self):
        self.require(not self.open_files, "file not closed")
```

Fig. 8: FileSystemMonitor class generated by the Gemini 2.5 LLM.

described where an LLM is used to generate a monitoring framework for a user defined logic, and subsequently used to generate monitors in that framework for user provided properties in that logic.

9 Conclusion

We have discussed different formalisms for formulating monitors, ranging from external DSLs, requiring their own grammar and parser, over internal DSLs which are libraries in a general-purpose programming language, and hybrid DSLs, which combine external and internal DSLs, to using a general-purpose programming language without the use of any special RV libraries. We have shown five solutions covering this spectrum. We have discussed the advantages and disadvantages of each approach. External DSLs allow succinct specifications and are easy to analyze but usually are less expressive. Internal (shallow) DSLs are very expressive (Turing complete). They e.g. allow performing data analysis, producing data results beyond just Boolean verdicts. However, specifications are usually more verbose, and they are harder to analyze and optimize. Hybrid DSLs attempt to achieve the advantages of both while minimizing the disadvantages. One can also "just" program monitors in a general-purpose programming language. This is probably how many monitors are currently developed in industry. This approach is less attractive if there are many, or evolving, requirements to be monitored. We illustrated how LLMs can perhaps address this issue by generating monitors from natural language requirements.

References

1. Bernhard Aichernig and Klaus Havelund. Correct-ish by design: From upfront verification to continuous monitoring of LLM generated code. In Bernhard Steffen, editor, *AISoLA 2024: Bridging the Gap Between AI and Reality.* Springer, 2025.
2. Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In Antti Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.
3. 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.
4. 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.
5. 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.
6. Itay Cohen, Klaus Havelund, Doron Peled, and Yoav Goldberg. The power of reframing: Using LLMs in synthesizing RV monitors. In Bettina Könighofer and Hazem Torfah, editors, *25th International Conference on Runtime Verification (RV)*, LNCS. Springer International Publishing, 2025. To appear.
7. 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.
8. 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.

9. 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.
10. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transf.*, 18(2):205–225, 2016.
11. DejaVu tool source code. https://github.com/havelund/dejavu.
12. Bevin Duckett, Klaus Havelund, and Luke Stewart. Space telemetry analysis with PyContract. In Anne E. Haxthausen, Wen-ling Huang, and Markus Roggenbach, editors, *Applicable Formal Methods for Safe Industrial Products - Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday*, volume 14165 of *LNCS*. Springer International Publishing, 2023.
13. Andy Dustman. Python MySQL. https://pypi.org/project/MySQL-python/, 2024.
14. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.
15. Gemini large language model. https://gemini.google.com.
16. 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.
17. Sylvain Halle and Roger Villemaire. Runtime enforcement of web service message contracts with data. volume 5, pages 192–206, 2012.
18. 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.
19. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, 17(2):143–170, 2015.
20. Klaus Havelund. Daut - Monitoring Data Streams with Data Automata. https://github.com/havelund/daut, 2024.
21. 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.
22. 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.
23. 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.
24. 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.

25. Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
26. Top programming languages 2024. https://spectrum.ieee.org/top-programming-languages-2024.
27. Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. https://en.wikipedia.org/wiki/Yacc.
28. 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.
29. 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.
30. Y. Matiyasevich. *Hilbert's 10th Problem.* MIT Press, 1993.
31. Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.
32. PyContract. https://github.com/pyrv/pycontract.
33. PyDejaVu tool source code. https://github.com/moraneus/pydejavu.
34. Jean-Pierre Queille and Joseph Sifakis. Iterative methods for the analysis of petri nets. In Claude Girault and Wolfgang Reisig, editors, *Application and Theory of Petri Nets, Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets, Strasbourg, France 23.-26. September 1980, Bad Honnef, Germany, 28.-30. September 1981*, volume 52 of *Informatik-Fachberichte*, pages 161–167. Springer, 1981.
35. Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: Monitoring at runtime with QEA. In Christel Baier and Cesare 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.
36. 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.
37. Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 133–191. Elsevier and MIT Press, 1990.
38. TP-DejaVu tool source code. https://github.com/moraneus/TP-DejaVu.
39. Windsurf editor. https://windsurf.com.