Noname manuscript No.
(will be inserted by the editor)

A Taxonomy for Classifying Runtime Verification Tools

Yliès Falcone · Srđan Krstić · Giles Reger · Dmitriy Traytel

the date of receipt and acceptance should be inserted later

Abstract Over the last 20 years Runtime Verification (RV) has grown into a diverse and active field, which has stimulated the development of numerous theoretical frameworks and practical tools. Many of the tools are at first sight very different and challenging to compare. Yet, there are similarities. In this work, we classify RV tools within a high-level taxonomy of concepts. We first present this taxonomy and discuss its different dimensions. Then, we survey the existing RV tools and, where possible with the support of tool authors, classify them according to the taxonomy. While the classification continually evolves, this article presents a snapshot with 60 state-of-the-art RV tools. We believe that this work is an important step in establishing a common terminology in RV and enabling a meaningful comparison of existing RV tools.

1 Introduction

Runtime Verification (RV) [24, 76, 78, 115] (or runtime monitoring) is (broadly) the study of methods to analyse the dynamic behaviour of computational systems. The most typical analysis is to check if a given run of a system satisfies a given specification, otherwise, when possible, alter the run such that it does. It is this general setting (and its variants) that we consider in this article. Whilst topics such as *specification mining* or *trace visualisation* are generally considered to be within this broad field, we do not include them in our discussion.

This article presents a taxonomy of RV frameworks and tools and uses this taxonomy to classify a large number of existing tools. It thereby extends an earlier work presented at the RV 2018 conference [79]. Specifically, this article expands and refines a number of points in the taxonomy and provides additional explanations. It also significantly extends the classification from 20 to 60 tools. Most newly added tools were classified by the tool authors themselves as part of a comprehensive survey within the RV community, which we have conducted.

This work is timely for a number of reasons. Firstly, after more than 15 years of maturing, the field has reached a point where such a general view is needed. The last significant attempt at a taxonomy was in 2004 [64] and had a distinctly different focus to our own. Secondly, a number of activities, such as the runtime verification competitions [21, 23, 81, 139], the RV-CuBES workshop [137, 140], two schools dedicated to RV [54, 77], and a COST action [1], which included the development of a tutorial book on the topic [22], have put the development of runtime verification tools into focus.

Contributions. This article has two main contributions:

- We present a detailed *taxonomy* that defines seven major concepts used to classify RV approaches (Section 3). Each concept is refined and explained, with areas of possible further refinement identified.
- We carry out a survey among the RV tool authors that results in the *classification* of 60 tools according to our taxonomy (Section 4). This both extends and subsumes the previous classification [79], which focused on only 20 tools and was performed without involving the tool authors.

We then discuss what we have learned from these two activities (Section 5) before concluding with some comments on how we see this work developing in the future (Section 6).

Y. Falcone
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, Laboratoire d'Informatique de Grenoble, 38000 Grenoble, France
E-mail: ylies.falcone@univ-grenoble-alpes.fr

S. Krstić
Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
E-mail: srdan.krstic@inf.ethz.ch

G. Reger
University of Manchester, Manchester, UK
E-mail: giles.reger@manchester.ac.uk

D. Traytel
Department of Computer Science, University of Copenhagen, Denmark
E-mail: traytel@di.ku.dk

2 A Brief Introduction to Runtime Verification

The field of RV is broad and the used terminology is not yet unified, although there have been attempts to standardise some concepts [24, 78, 115]. One issue with terminology is that it often fixes some part of the taxonomy which we introduce shortly. For instance, terminology may assume something about the role of a particular component or its relation with another component. For the sake of clarity within this article, we fix the following terms at a relatively abstract level:

- *Monitored system.* The system consisting of software, hardware, or a combination of the two, that is being monitored. Its behaviour is usually abstracted as a *trace*.
- *Initial system.* A version of the monitored system that is not monitored and thus not affected by monitoring.
- *Trace.* A finite sequence of observations that represents (or in some cases approximates) the behaviour of interest in the monitor system. The process of extracting/recording the trace is usually referred to as *instrumentation*.
- *Property.* A partition of the set of all traces. This may simply be a separation of traces into two sets or a more refined classification of traces.
- *Specification.* A concrete description of a property using a well-defined formalism.
- *Monitor.* A runtime object that is used to check properties. The monitor will receive observations from the trace (usually incrementally) and may optionally send information back to the monitored system, or to some other source.
- *RV framework.* A collection consisting of a specification formalism, monitoring algorithm(s) (for generating and executing monitors), and (optional) instrumentation techniques that allows for runtime verification.
- *RV tool.* A concrete instance of an RV framework.
- *Overhead.* Any form of performance penalty sustained by the monitored system and caused by the RV tool.

To make these terms more concrete, without committing to a particular interpretation, we present two illustrative example scenarios that will be used throughout the following section.

Example 1 (File system) Consider the setting of ensuring that the usage of a file system is correct. The *monitored system* may be the file system itself or it may be an application using the file system. Either system could be *instrumented* to record a sequence of file operations as discrete events to form a *trace*. Events would contain information about the name of each executed file operation as well as additional metadata such as the relevant file handler, the user, access permissions and so on. *Properties* of interest would include ensuring that files are opened before being read, that opened files are eventually closed, and that files are accessed with the correct permissions.

Example 2 (Hybrid engine) Consider the setting of monitoring the behaviour of a hybrid engine in an automobile. The *monitored system* is the embedded system, which itself consists of a combination of distributed software and hardware components. The *trace* may be a set of continuous *signals* recorded at different locations within the monitored system. The monitor is itself an embedded system and may either be centralised or distributed. *Properties* of interest would include checking that the electric power does not fall below a certain threshold for more than a given period, the average fuel consumption is within a given range, or the shift between engine modes occurs within a certain delay of being triggered.

Later, we will see examples of how properties for each example may be specified and monitored.

3 A Taxonomy of Runtime Verification

This section describes a taxonomy of runtime verification. Figure 1 provides a general overview of the taxonomy which identifies the seven major concepts (and is limited to the first two levels for readability reasons). This taxonomy provides a hierarchical organisation of the major concepts used in the field.

Development process. This taxonomy was developed in an iterative process alongside the classification presented in Section 4. The seven main conceptual areas were identified as an initial starting point and extended with established dichotomies (e.g., offline vs online monitoring). Sub-concepts were then added and refined based on the focused classification process and a wider survey of tools (involving over 50 tools, not described in this article). We have attempted to ensure that the taxonomy remains as general and flexible as possible.

Relations between nodes. We do not capture concepts such as mutual exclusion or interdependence between nodes graphically but aim to describe these in the text. In most cases the final level of the taxonomy captures some concrete instances of a particular (sub-)concept and it is at this level where such relations are most important.

The remainder of this section focuses on each of the seven major concepts and expands the description along the corresponding branches.

3.1 Specification

The specification part of the taxonomy is depicted in Figure 2. A specification describes the intended system behaviour (property), that is *what one wants to check* when monitoring the system. It is one of the main inputs to an RV framework and it is formulated before running the system.

A specification exists within the context of a general system model i.e., the abstraction of the monitored system. The main purpose of such a model is to define the information that can be obtained by observing the monitored system (see Section 3.5), but it may also define other contextual information.

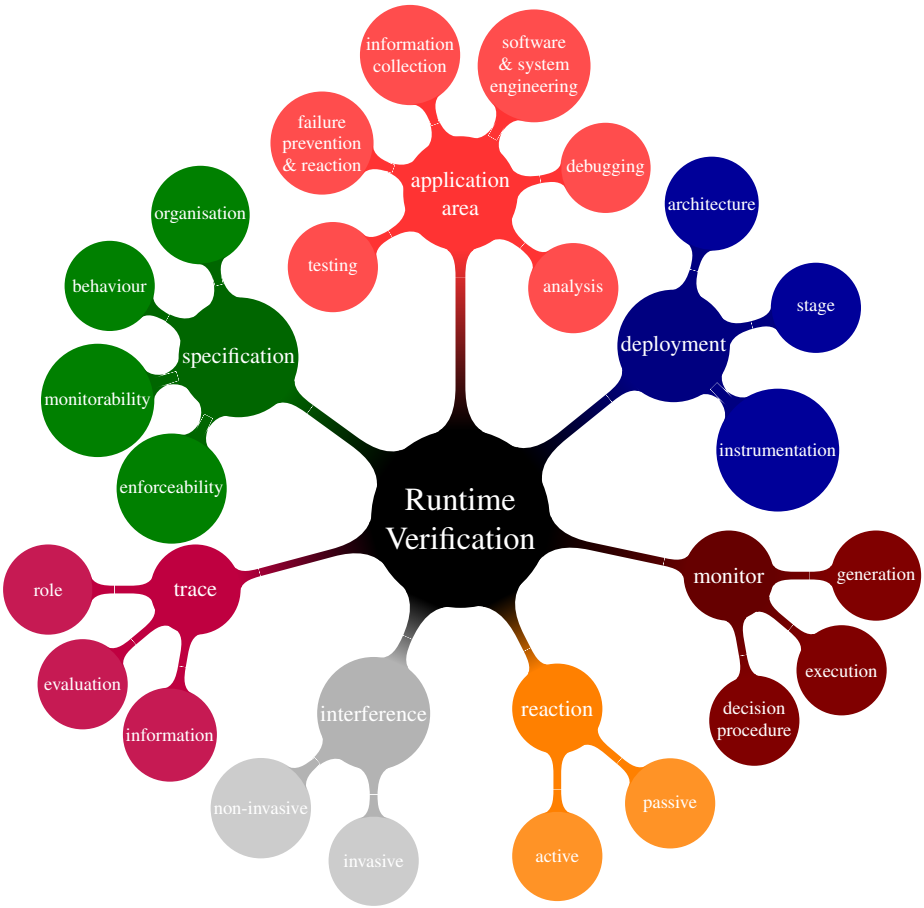


Fig. 1: Mind map overviewing the taxonomy of Runtime Verification

A specification itself can be **organised** in a centralised or decentralised fashion, in relation to the system being monitored. **Centralised** specifications are more common; they are monolithic descriptions of the intended system behaviour and abstract away from the system architecture. **Decentralised** specifications are organised in interdependent modules; their organisation can follow the monitored system's architecture or some other logical structure [72, 74, 85, 114, 130, 131, 146].

Moreover, a specification can be either **implicit** or **explicit**, depending on the desired **behaviour** to be monitored.

Implicit specifications. An **implicit** specification is used in an RV framework when there is a general understanding of the particular desired behaviour. RV tools do not require their users to explicitly formulate and enter implicit specifications. Implicit specifications generally aim at avoiding runtime errors/violations (that typically would not be caught by a compiler or before the system deployment). Such runtime errors can be critical. Three categories of implicit specifications can be distinguished. First is (memory) safety [153, 156], whose purpose is to ensure proper accesses to the system memory by avoiding accesses to undefined memory. Safety precludes memory errors [162] such as use after free, buffer over-reads and overflows, null pointer dereferences, as well as divisions (or moduli) by zero, arithmetic overflows, incorrect downcasts and coercions, uncaught exceptions, etc. Second is correct concurrent behaviour which aims at guaranteeing absence of deadlocks, the atomicity of operations, the absence of data races, missed signals and order violations [116]. Third is system security which revolves around specifying flavors of integrity, confidentiality, and availability. The final layers of implicit specification sub-concepts shown in Figure 2 are non-exhaustive lists of prominent examples.

Explicit specifications. An **explicit** specification is one provided by the user of an RV framework and formally expresses a monitored system's property. It can complement the properties checked by the compiler of a language (e.g., errors that would not be caught by type checking). An explicit specification denotes a function from traces to some **output** domain (discussed below), which encodes the classification of traces as defined by the specified property. An explicit specification is written in the RV framework's specification formalism, which belongs to some **paradigm**. For instance, specifications may describe the above-mentioned function **operationally** (e.g., as a finite-state automaton) or **declaratively** (e.g., as a temporal logic formula). The specification formalism can

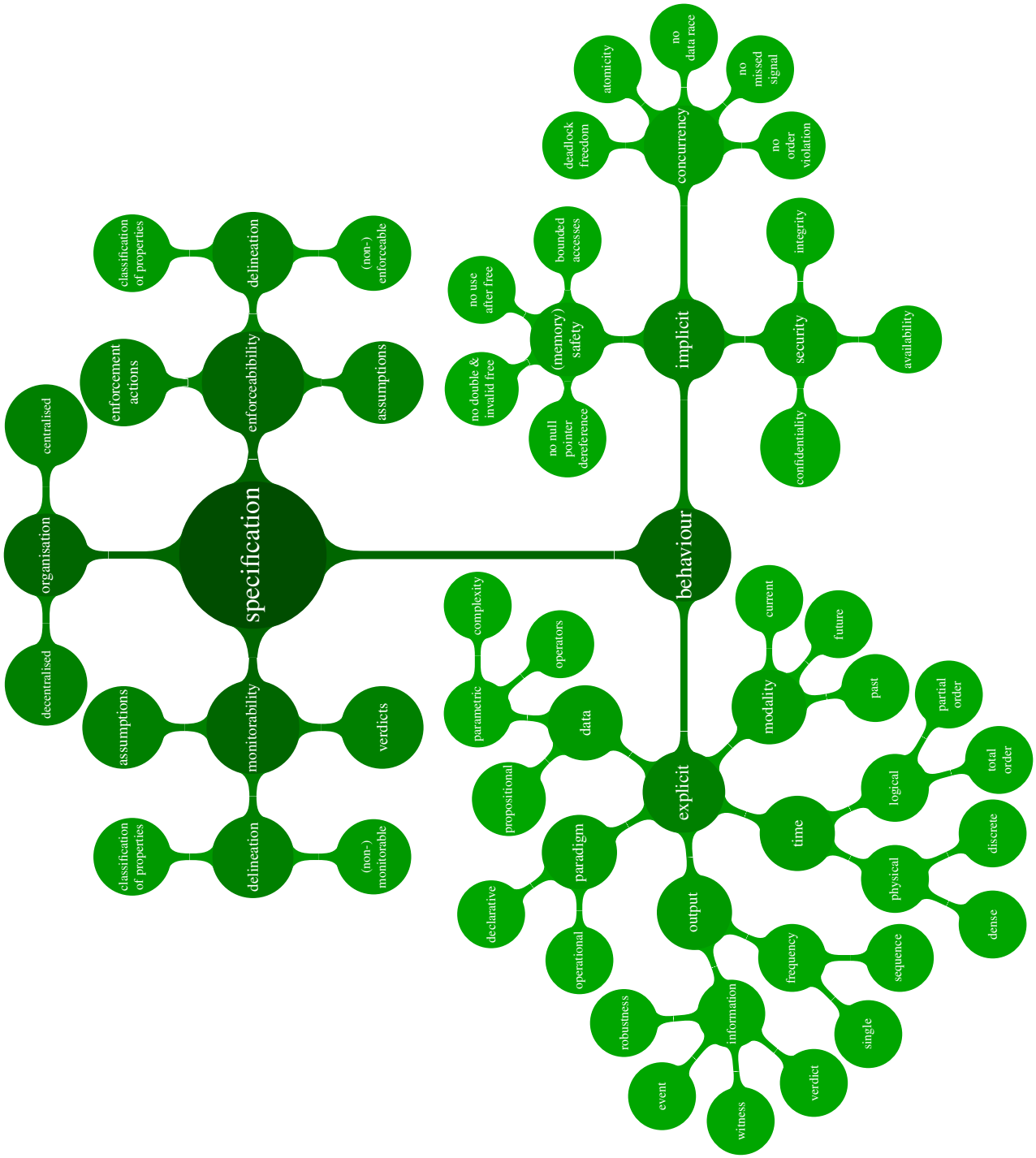
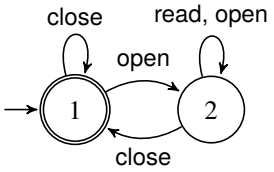


Fig. 2: Mind map for the specification part of the taxonomy

offer different features used to model the expected behaviour according to the dimensions discussed below.

The specification formalism may support different **modalities**. Some formalisms restrict assertions to the **current** observation whereas others support assertions over the **past** or **future** observations. In some cases, these different modalities are crucial for the formalism's expressiveness; in other cases they only improve usability (e.g., by improving conciseness).

Example 3 (Specification paradigm and modality) For the file system from Example 1, we consider the simple property: *a file must be opened before being read and must be closed before the end of the program*. The property may be formally expressed as below by a finite state automaton (left) or in linear temporal logic (right), assuming that the system model is a trace consisting of individual file operations.



$$\Box\Big((\mathsf{read} \rightarrow (\neg \mathsf{close}\ \mathcal{S}\ \mathsf{open})) \wedge (\mathsf{open} \rightarrow \Diamond\ \mathsf{close})\Big)$$

An automaton is *operational*, it describes the operations that should be taken to process the input to compute the output. The states capture information about the *past* and the transitions capture requirements about *future* actions/requirements. As such, automata can be considered as capturing a *past implies future* modality scheme. Automata that process the input in the reverse order capture the dual *future implies past* modality.

The temporal logic formula uses a past-time operator $\mathcal{S}$ to state that if read is observed then in the past the file should have been opened and not closed in the meantime. The formula also uses the future-time operator $\Diamond$ to capture the requirement that the file is eventually closed. As the specification simply declares what should happen (and not how to check this) the approach is *declarative*.

A key dimension is how specifications or a specification formalism handle data in observations. The simple case is when observations are **propositional**, i.e., they are assumed to be atomic and unstructured (e.g., simple names). Otherwise, we say that the approach is **parametric**: observations are associated with a list of (possibly named) runtime values, called *parameters*. The structure of these parameters may have different **complexity**, e.g., they may be simple primitive values or values with a complex hierarchical structure, like XML documents or runtime objects. The specification formalism must support ways of predicating on the structure of complex parameters. The **operators** over parameters supported by the specification language may also vary. For example, whether it is possible to compare parameters in different ways (e.g., more than equality) or whether quantification (e.g., first-order, freeze quantification, or pseudo-quantification via templates) over parameters is supported [95, 97, 101].

Example 4 (Parametric specification) In Example 3, we saw a property over *propositions* indicating whether a file was opened, read, or closed. However, we may wish to identify different files and use *parametric* observations such as open(*readme.txt*), which contains a runtime value denoting the opened file name. The specification formalism may then introduce new operators such as quantification or counting over the parameters. One could then specify that *when opening a file f the file can be read at most m times (reset when the file is opened again), where the bound m is given when the file is opened* as shown below.

$$\forall f.\, \Box\,(\mathsf{open}(f, m) \rightarrow (\#[\mathsf{read}(f) \mid \blacklozenge\, \exists m'.\mathsf{open}(f, m')] \leq m))$$

The expression $\#[A \mid B]$ counts the number of occurrences of A over the (shortest) part of the previous trace making B true.

Example 5 (Operators over parameters) In Example 2 (Hybrid engine), we may have access to a real-valued signal ep that provides values of the current electric power. In our taxonomy, a real-valued signal can be seen as a special case of a parametric observation with a single real-valued parameter. In the trace part of the taxonomy (Section 3.5), we capture the distinction between signals represented as a (discrete) sequence of observations and as a (continuous) closed-form expressions.

In stream-based specification formalisms it is common to use functions on signals to transform them into Boolean signals, which indicate whether a property is satisfied by the system over time. Such functions are examples of the operators over parameters. For instance, we might specify that *electric power stays above a threshold E* simply as

$$\Box(\mathsf{ep} > E)$$

Alternatively, we may have a reference signal $\mathsf{ep}_{\mathsf{ref}}$ that we want ep to track with some allowed deviation (Δ), for example:

$$\begin{aligned} \mathsf{diff} &= \mathsf{abs}(\mathsf{ep} - \mathsf{ep}_{\mathsf{ref}}) \\ \mathsf{error} &= \mathsf{diff} > \Delta \end{aligned}$$

Note that the two specifications above differ in the way they formalise the respective properties. The former captures desirable behaviour, while the latter captures undesirable behaviour. Operators $>$, $-$, and abs are applied point-wise to the values of signals ep, $\mathsf{ep}_{\mathsf{ref}}$, and diff, as well as to the constant values E and Δ. The relation of such real-valued signals to the system and each other is discussed later (Section 3.5).

A specification can also express constraints over **time**. Such constraints refer either to **logical time** or **physical time**.

In the case of logical time, the specification describes the desired relative ordering between observations. Such an order can be **total** (e.g., when monitoring a monolithic single-threaded program) or **partial** (e.g., when monitoring a multi-threaded program or a distributed system).

In the case of physical time, the specification describes the desired physical time that must elapse between the observations of a running the system. The observations are then associated with physical time (called *timestamps*), the domain of which can be **discrete** or **dense**.

To exploit the properties of physical time (e.g., monotonicity) to improve performance, RV tools typically implement special provisions to monitor constraints over physical time. However, there is a special case where some tools treat physical time as any other parameter associated with observations. Such approaches typically do not offer specification formalisms with native support for expressing physical time constraints, but rather rely on the operators over parameters.

Example 6 (Logical and physical time) For Example 1 (File system), let us consider the property that *a file should only be read if it was opened in the last five minutes*. We might specify this in a past-time metric temporal logic where temporal operators are extended with time intervals. For example

$$\square\left(\mathsf{read} \rightarrow (\neg \mathsf{close}\ \mathcal{S}_{[0,5]}\ \mathsf{open})\right)$$

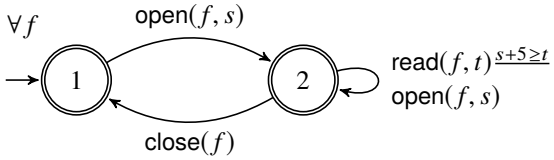
should be read as *it is always the case that if a* read *occurs then at some point in the past 5 time units a* open *occurred with no* close *occurring in-between*. Note that the specification assumes that the granularity of the physical time is in minutes.

In contrast to the specifications in Example 3, which express constraints over logical time, the specification in this example expresses a constraint over physical time.

Example 7 (Time as data) The property from Example 6 can also be expressed using the *time as data* paradigm where events are extended with timestamps. For example

$$\forall f, s, t.\ \square\left(\begin{array}{l}\mathsf{read}(f,t) \rightarrow \\ \quad (\neg \mathsf{close}(f)\ \mathcal{S}\ (\mathsf{open}(f,s) \wedge s + 5 \geq t))\end{array}\right)$$

introduces two timestamps (t and s) as well as quantifying over the file being operated on (f). The *time-as-data* paradigm is more common in some operational languages, such as the following quantified automaton, where time is more difficult to incorporate into standard operators.



This automaton states that, for each file, the system can be in one of two safe states: either the file is closed and cannot be read; or it is open and each read must occur within 5 minutes of the previous open event.

Example 8 (Dense time) Extending Example 5, we may want to allow ep to track $\mathsf{ep}_{\mathsf{ref}}$ with some delay, i.e., the signal may deviate from the reference for a brief period of time.

Consider specifying that *electric power does not fall below the threshold E for more than a given time period* ϵ.

$$\begin{aligned}\mathsf{fall} &= \mathsf{ep}(t) < E \\ \mathsf{recover} &= \mathsf{ep}(t+\epsilon) \geq E \\ \mathsf{error} &= \mathsf{fall} \wedge \neg \mathsf{recover}\end{aligned}$$

Here, signals are indexed using physical time. Signals in Example 5 are indexed implicitly with the same time.

Operational languages for specifications over dense time also include timed automata and their extensions to handle data. Such automata are extended with a (finite) set of real-valued clocks that can be checked and reset along transitions.

The last dimension of an explicit specification concept is the **output** assigned to the traces. Specifically, it is the codomain of the function fixed by the explicit specification.

In the standard case, the specification associates **verdicts** to a trace. The verdicts indicate specification satisfaction or violation and may range over a domain extending the Boolean domain. A more refined output includes a **witness** in addition to the verdict, e.g., a set of bindings of values to free variables in the specification that lead to violations. Another form of refinement is to output **events**, which may be created or come from the execution. When events are created, these are aggregation or fusion of existing (information) events, while when events come from the execution these are typically important events witnessing violation or satisfaction of the specification. **Robustness** information extends classical verdicts by providing a quantitative assessment of the (degree of) specification satisfaction or violation. Finally, the output **frequency** can be a **single** piece of information or a **sequence** of information. For instance, the standard semantics of temporal logic defines formula satisfaction at any point in an infinite word. Hence, such specifications may denote functions from words to *sequences* of verdicts. Note that for specifications in general the output may be a sequence of some arbitrary information (e.g., witnesses). These are computed based on the specification and the data from the observations.

Example 9 (Verdicts) Given the property and its specification in Example 6, we might observe a timed trace

$$\mathsf{open}_0\ \ \mathsf{read}_3\ \ \mathsf{read}_6\ \ \mathsf{close}_7\ \ \mathsf{read}_8\ \ \mathsf{close}_9\ \ \mathsf{open}_{12}\ \ \mathsf{read}_{14}$$

that violates the property. Note that besides having a file operation name, each observation also has a timestamp.

The output may simply be to report that the trace violates the specification. It may additionally identify the 3rd event as the (first) *witness* of the violation. Another approach might be to quantify the violation by measuring the distance between the given trace and an accepting one based on some established distance measure. In this case, two events (the 3rd and 5th) should be altered to get a satisfying trace. Finally, we may want a *sequence* of verdicts each stating if the formula is satisfied at a point in the trace. This effectively amounts to

evaluating the subformula ϕ of a formula $\Box\,\phi$ at each point in the trace. In this case, the stream would be:

true true false true false true true true

which is straightforward to compute if ϕ is a past-time formula. For bounded future formulas the result is potentially delayed, while for arbitrary future formulas the definitive true or false verdicts may not be known without inspecting an infinite trace. To address this, some specifications refine the set of verdicts to {true, false, ?}, where ? indicates an inconclusive result. Further refinements of this set also exist.

Example 10 (Witnesses) Given the property and its specification in Example 7, we expect a trace with time encoded as an additional parameter in some observations.

$\mathsf{open}(f_1, 0)$ $\mathsf{read}(f_1, 3)$ $\mathsf{read}(f_1, 6)$ $\mathsf{read}(f_2, 8)$

When monitoring this trace, the output may look as follows:

true true false$(f \mapsto f_1, s \mapsto 0, t \mapsto 6)$ false$(f \mapsto f_2, t \mapsto 8)$.

In particular it includes bindings of variables f, s, and t from the specification as *witnesses* for each violation in the trace.

Finally, the two last concepts related to specifications in general are their **monitorability** and **enforceability**. At an abstract level, a specification is said to be **monitorable** if it is worth runtime verifying such specification because a verdict can be reached eventually. While a specification is said **enforceable** if such specification can be enforced on the system. Both monitorability and enforceability rely on various assumptions, e.g., on the faithfulness of the observations or on the effectiveness of the **enforcement** actions used to modify the initial system behaviour. We refer to [24] and [76, 80] for overviews of monitorability and enforceability, respectively.

3.2 Monitor

The monitor part of the taxonomy is depicted in Figure 3. A monitor is a main component of a runtime verification framework. The monitor is a component that performs checks on the execution of a system as prescribed by the specification. The checks can be seen as decision problems solved by a decision procedure, which the monitor implements.

A **decision procedure** implemented by a monitor can be either **analytical** or **operational**. Analytical decision procedures query and scan information about the observations (e.g., from a database) to determine whether some condition holds in the current execution. Operational decision procedures are those based on automata or formula rewriting. In an **automata-based** decision procedure, the monitor relies on some automata-like formalism (either classical finite-state automata or richer variants) to perform the required checks. The **rewrite-based** decision procedure is based on a set of (possibly predefined) rewrite rules triggered by a new event.

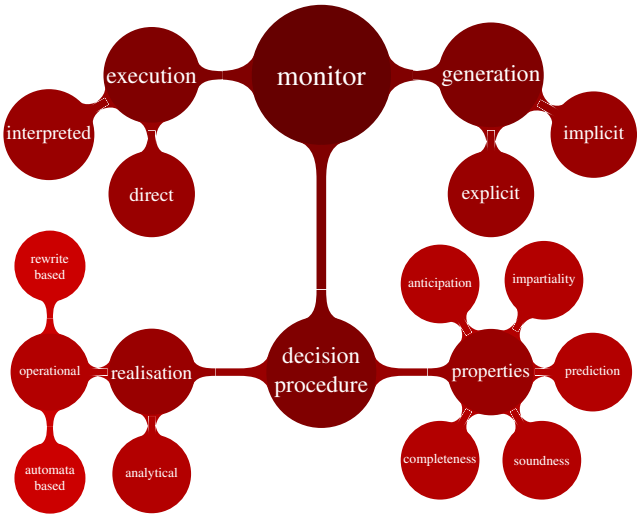


Fig. 3: Mind map for the monitor part of the taxonomy

Example 11 (Decision Procedures) To contrast the different forms of decision procedures, we consider different monitoring algorithms for a form of first-order LTL and a new property in the setting of Example 1: *every file that is written to is eventually saved and then closed*. We might specify this as

$$\psi \stackrel{\text{def}}{=} \forall f.\, \Box\,(\mathsf{write}(f) \longrightarrow \Diamond(\mathsf{save}(f) \wedge \Diamond\,\mathsf{close}(f)))$$

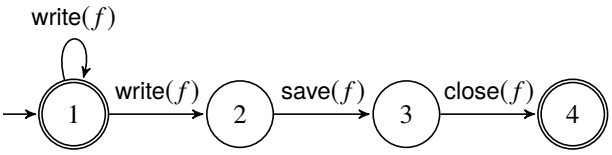
There are then three approaches to monitoring that we exemplify on the following simple trace

$\mathsf{write}(A)$ $\mathsf{save}(A)$ $\mathsf{write}(B)$ $\mathsf{close}(A)$ $\mathsf{close}(B)$

which does not satisfy the property as file B is not saved.

An **analytical** approach might store the trace as a *temporal database*, e.g., a sequence of tables where each table gives the interpretation of an event at a current time point. We can then identify which files violate the property by interpreting the logical operations using operations from relational algebra. In this case, we would incrementally perform a *union* with the write table and *antijoin*s with the (union of the) save and close tables. Here the result would be table consisting of a single line for the B file. While approaches that work with finite tables are limited to a safety-fragment in which queries guarantee finite output, there also exist generalisations that operate on finite representations of infinite relations [31, 99].

An **automata-based operational** approach might generate the following non-deterministic automaton where variables are bound on their first match.



Evaluating the trace on this would identify a final set of configurations $\{\langle 4, [f \mapsto A]\rangle, \langle 2, [f \mapsto B]\rangle\}$. The existence

of a non-accepting state in this configuration would indicate that the trace does not satisfy the property.

A **rewrite-based operational** approach might directly rewrite the formula using the trace as follows

$$\begin{aligned}
\psi &\overset{\text{write}(A)}{\Longrightarrow} \psi \wedge \Diamond(\text{save}(A) \wedge \Diamond\, \text{close}(A)) \\
&\overset{\text{save}(A)}{\Longrightarrow} \psi \wedge \Diamond\, \text{close}(A) \\
&\overset{\text{write}(B)}{\Longrightarrow} \psi \wedge \Diamond\, \text{close}(A) \wedge \Diamond(\text{save}(B) \wedge \Diamond\, \text{close}(B)) \\
&\overset{\text{close}(A)}{\Longrightarrow} \psi \wedge \Diamond(\text{save}(B) \wedge \Diamond\, \text{close}(B)) \\
&\overset{\text{close}(B)}{\Longrightarrow} \psi \wedge \Diamond(\text{save}(B) \wedge \Diamond\, \text{close}(B))
\end{aligned}$$

As the final formula is not satisfied by the empty trace, the trace does not satisfy the property.

When designing monitors, it is desirable that its decision procedure guarantees several properties. Intuitively, a **sound** monitor never provides incorrect output, while a **complete** monitor always provides an output. The properties reflect how much confidence one can have in the output of monitor and how much confidence one can have that a monitor will produce an output, respectively.

Soundness and completeness cannot be guaranteed in situations where, for instance, some form of sampling is used and thus not all observations are received by the monitor. Similarly, when the order of observations as received by the monitor does not match the execution order. In such cases, the monitor can perform two kinds of **prediction**. Firstly, a monitor may use a model of the monitored system to predict the possible future system executions, evaluate all of them, and output a prediction. Secondly, a monitor may predict potential errors in alternative concurrent executions (which are not actually observed by the monitor).

A monitor is **impartial** when the produced outputs are not contradictory over time. Finally, a monitor can **anticipate** the output. This resembles prediction but the knowledge used by the monitor in this case comes only from the monitored specification. Impartiality and anticipation are properties of the semantics of the specification language itself, which may or may not be realised by the given decision procedure. For example, a language may allow anticipation but a given decision procedure may not realise this property.

Example 12 (Impartial and anticipatory verdicts) To demonstrate the different properties of a decision procedure consider the property given in Example 3 that *a file must be opened before being read and must be closed before the end of the program*. A decision procedure that returns a *false* verdict when a file has been opened but not closed (e.g., from state 2 in the automaton given in Example 3) would not be *impartial* as the verdict could become *true* on observing a file close event. A decision procedure that returns a *false* verdict when a file is read without being opened correctly *anticipates* that no future events can change the outcome.

The monitor may be **generated explicitly** from the specification (e.g., an automaton synthesised from an LTL formula) or may exist **implicitly** (e.g., a rewrite system defined in an internal domain-specific language). Finally, a monitor must be **executed**. This might proceed **directly** if the monitor is given as code, e.g., when the monitor is either already implemented as some extension of a programming language (i.e., an internal domain-specific language), or the synthesis step from generation directly produced executable code. Otherwise, the monitor is said to be **interpreted**. The key difference between the two approaches is whether each monitor is implemented by a separate piece of code (direct) or there is a generic monitoring code that is parametrised by some monitor information (interpreted). Fig. 4 illustrates the different approaches to generation and execution.

Example 13 (Generation and execution) To illustrate the different notions of generation and execution, let us return to the approaches discussed in Example 11. In the automata-based operational approach we had to *generate* the automaton from the logical specification but in the other approaches the monitor (e.g., the structure required to evaluate the trace) was implicit. In all cases, we could view monitoring as *interpreting* the monitor over the data. But the automaton-based approach could have been further compiled down into code that directly computed the next configurations without the need for interpretation.

To make the different approaches illustrated in Fig. 4 concrete we provide examples of tools (from the classification, see Section 4) that fit into each category. DejaVu [98] explicitly generates Scala monitors from FO-PLTL formulas and then directly executes these monitors (as programs) on traces. Eagle [16] explicitly generates monitors (from fixed-point temporal logic formulas) as in-memory objects that are then interpreted by the (rewriting-based) monitoring algorithm. Monitors in BeepBeep [92] are implicitly constructed from a library and then executed directly. Monitors in MonPoly [31] are implicitly defined as formulas that are then interpreted as queries.

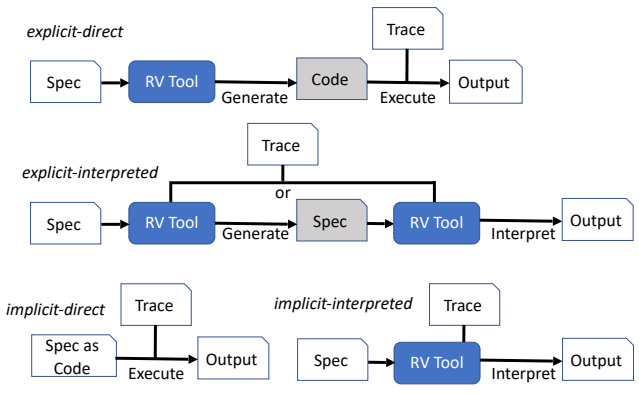


Fig. 4: The different generation and execution approaches.

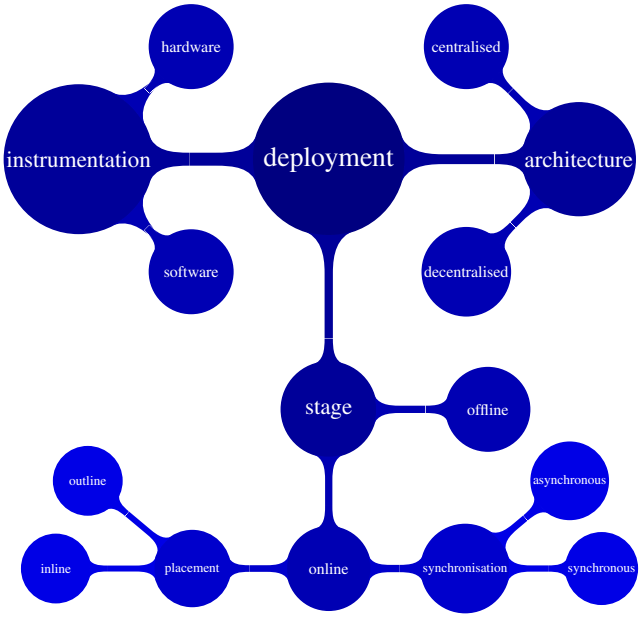


Fig. 5: Mind map for the deployment part of the taxonomy

3.3 Deployment

The deployment part of the taxonomy is depicted in Figure 5. By deployment, we refer to how the monitor is effectively implemented, organised, how it retrieves the observations from the system, and when it does so.

The notion of **stage** describes *when* the monitor operates, with respect to the execution of the system. Runtime verification is said to apply **offline** when the monitor runs after the system finished executing and thus has access to the complete system execution (e.g., a log file). It is said to apply **online** when the monitor runs while the system executes and thus observes the current execution and a part of its history. In the online case, the communication and connection between the monitor and the system can be **synchronous** or **asynchronous**, respectively depending on whether the monitored system stops executing while the monitor analyses the retrieved observation. It is possible for a monitor to be *partially* synchronous if it synchronises on some but not all observations.

The notion of **placement** describes where the monitor operates in reference to the monitored system. Therefore, this concept only applies when the stage is online. Traditionally, the monitor is said to be **inline** (resp. **outline**) when it executes in the same (resp. in a different) address space as/than the monitored system.

Pragmatically, the difference between inline and outline is a matter of **instrumentation**. An inline tool implicitly includes some form of instrumentation, used to extract observations and inline the monitor in the monitored system.

In contrast, outline tools typically provide an interface for receiving observations. Such an interface may be exposed the monitor in the same programming language as used by the monitored system, which would then just call it directly. Otherwise, the interface would be invoked via some communication middleware (e.g., OS pipes or RPCs). There is a grey area between the two in the instance of tools that provide an outline interface but may also automatically generate instrumentation code.

Instrumentation may be at the **hardware** or **software** level with further subdivisions of these concepts covered in Section 4.

Example 14 (Instrumentation) To highlight two different approaches to instrumentation, let us consider the instrumentation required for our two running examples. In the file system setting (Example 1), our instrumentation will be at the software level. If we are monitoring a Java program to ensure it uses files properly we might *inline* our monitors via tools such as AspectJ [113] (instrumentation at the source level) or BISM [154] (instrumentation at the bytecode level) meaning that the monitored program and monitors run within the same Java Virtual Machine. In the hybrid engine setting (Example 2), instrumentation is likely to be at the hardware level with specific circuitry employed to relay signals to a the monitor, which will necessarily be *outline*.

Lastly, the architecture of the monitor may be **centralised** (e.g., deployed as one monolithic component) or **decentralised** (e.g., deployed as multiple synchronously or asynchronously communicating components).

Example 15 (Deployment) We provide examples of tools (from the classification, see Section 4) that fit into various categories with respect to deployment. The detectER [50] tool runs asynchronously alongside a monitored Erlang program (e.g., online) placing monitors outline with instrumentation occurring via Virtual Machine tracing. Conversely, JavaMOP [119] uses AspectJ to weave generated monitors into Java programs (placing them inline) to be executed online synchronously. Finally, tools such as Hydra [135] and DecentMon [36] operate offline on collected traces.

3.4 Reaction

The reaction part of the taxonomy is depicted in Figure 6. By reaction, we refer to the activity that the monitor performs after checking the specification. It may actively affect the execution of the monitored system, or passively report information.

Reaction is said to be **passive** when the monitor does not influence[1] the execution of the monitored system. A passive monitor is typically an observer, only collecting information. This means that there are assumptions and/or guarantees that the analysis performed by monitor did not alter the execution and thus the reported information is accurate (e.g., to ensure

[1] We note that the total absence of influence is impossible because of the need for instrumentation; see Section 3.6.

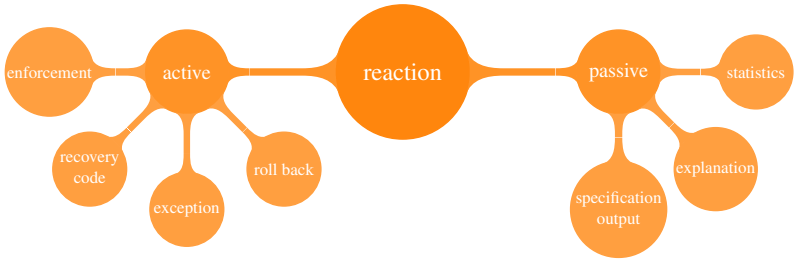


Fig. 6: Mind map for the reaction part of the taxonomy

the soundness of verdicts by the absence of false positive and negatives). Examples of guarantees include a form of behavioural equivalence (e.g., simulation or bisimulation, or their weak variants) between the initial system and the monitored system.

Passive monitors typically produce the **specification output** (i.e., the elements of the codomain of the function denoted by the explicit specification). Alternatively, they may provide an **explanation** of a specification output (e.g., a part of the trace containing observations that lead to the specific output) or **statistics** (e.g., the number of violated/satisfied specifications, or the number of times the inconclusive verdict ? was output before a conclusive verdict is reached) about a specification output.

Reaction is said to be **active** when the monitor affects the execution of the monitored system. Active reaction also encompasses the cases where the monitor modifies the output of the monitored system (e.g., by modifying the computed results, reducing or augmenting the output information).

Active reaction is only possible when the stage of monitoring is online (see Section 3.3), i.e., the monitor executes along with the monitored system. An active monitor would typically affect the execution of the system when a violation is detected. Various active reactions are possible. A so-called **enforcement** monitor can try to prevent property violations from occurring by forcing the system to adhere to the specification. When a violation occurs, a monitor can execute **recovery code** to mitigate the effect of the fault and let the program either terminate or pursue the execution from a safer state. A monitor can also raise **exceptions** that were already present in the initial system. Finally, the monitor can launch mechanisms that **roll back** the system to its latest correct state.

3.5 Trace

The trace part of the taxonomy is depicted in Figure 7. The notion of trace appears in two places in an RV framework and this distinction is captured by the **role** concept.

An **observed** trace is the sequence of observations extracted from the monitored system and examined by the monitor. Conversely the trace **model** is the mathematical object forming part of the semantics of the specification formalism. It corresponds to the system model concept in the specification part of the taxonomy (see Section 3.1). Clearly, RV frameworks must connect the two trace concepts, but it is important to state the properties that each concept has separately. For example, trace models may be **infinite** (as in standard LTL) whilst observed traces are necessarily **finite** – in such case the monitoring approach must evaluate a finite trace with respect to a property over infinite traces. A trace model must define the notions of **time** and **data**, present in the specification (see Section 3.1).

The extraction of the observed trace depends on the particular observation **sampling** technique and the **precision** at which the observations are made.

Sampling is said to be **event triggered** if the monitor receives an observation whenever some event of interest happens in the monitored system. Common events of interest are function calls/returns, relevant state changes, reception of input, or emission of output.

Sampling is said to be **time triggered** when there exists a fairly regular period at which observations are collected in the monitored system and sent to the monitor. The term sampling here reflects the fact that any trace will only collect a relevant subset of actual behaviours. If the monitoring tool assumes that the trace contains *all* relevant events then it is **precise**. Otherwise, it is **imprecise** and the tool must take the imprecision into account. Reasons for imprecision may be due to imperfect trace collection methods, or overhead reduction.

Both the observed trace and the trace model are abstractions of the monitored system's execution and can only contain some of the runtime information. For instance, the trace can contain information on the internal **state** of the program or notifications that some **events** occurred in the program (or both). Not exclusive of the previous option, the monitor can also process the **input and output** information from a transformational program[2]. Finally, the trace can contain time-continuous information in the form of a **signal**, which may be captured as a **closed-form expression** or by discrete sequence of **samples**.

The runtime information received by the monitor represents an **evaluation** of the monitored system's state. This

[2] A transformational program is a program that takes some input, processes it, delivers some output, and terminates (e.g., a compiler); as opposed to an interactive or a reactive program.

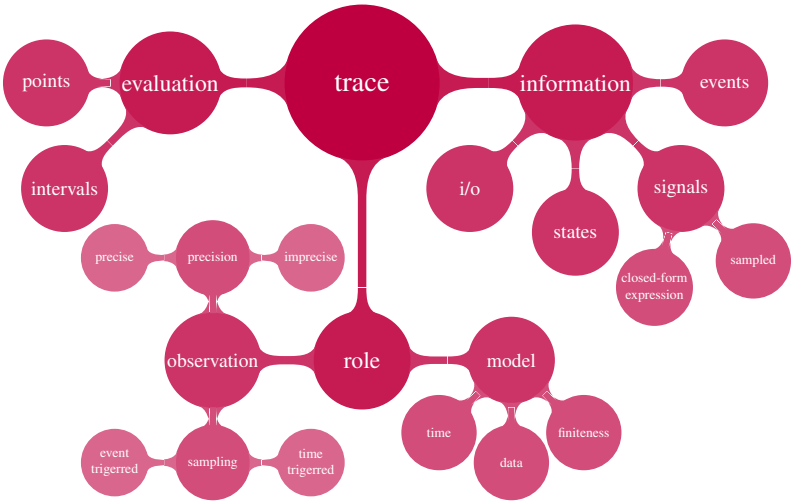


Fig. 7: Mind map for the trace part of the taxonomy

information can correspond to an identified **point** in time (or at a program location) or a time **interval**.

3.6 Interference

The **interference** part of the taxonomy (see Figure 1) characterises RV frameworks as **invasive** or **non-invasive**.

In general, a non-invasive RV framework is impossible due to the observer effect. However, in reality the level of monitor interference can be seen as a spectrum. The interference can amount to the induced overhead (time or memory wise) or by a modification of the RAM layout or CPU scheduling. There are two sources of interference of an RV framework with a monitored system. First, how much an RV framework interferes with the initial system depends on the effect of the instrumentation applied to the system, which itself depends on the specification as instrumentation is purposed mainly to collect a trace. Thus, the quantity of information in the trace and the frequency at which this information is collected (depending on the sampling) affects the degree of instrumentation. Moreover, interference also depends on the monitor deployment. Offline monitoring is considered to be less intrusive because the observation made on the system consists only in collecting observations into a trace; but it still requires a minimal form of instrumentation. Online monitoring is considered to be more intrusive to a degree depending on the coupling between the monitored system and the monitor. Second, interference with the monitored system also occurs when actively steering the system.

3.7 Application Areas

We have included **application areas** as a top-level concept of the taxonomy (see Figure 1), since it can have a large impact on other aspects of the RV tools. There are numerous application areas of runtime verification.

We have identified the following (non-exhaustive) categories. First, runtime verification can be used for the purpose of **collecting information** about a running system. This includes visualising its execution (e.g., with traces, graphs, or diagrams), evaluating its runtime performance using metrics (like execution time, memory consumption, communication, etc.), and collecting relevant statistics.

Second, runtime verification can be used to perform an **analysis** of a running system, usually to complement or in conjunction with static analysis. Such runtime analysis can focus on verifying the system (e.g., with respect to requirements, properties, or goals) to provide **security**, **privacy**, **safety**, and **progress/liveness** assurances.

Third, runtime verification can be used to augment software engineering techniques with a rigorous analysis of runtime information (e.g., by augmenting code coverage [3]).

Fourth, runtime verification can be used to complement other runtime techniques for finding defects and locating faults in systems such as **testing** (e.g., by augmenting unit testing [61]) and **debugging** (e.g., by augmenting interactive debugging [106]).

Finally, leveraging the previous techniques, runtime verification can be used to address the general problem of runtime **failure prevention and reaction**, by offering ways to detect faults, contain them, recover from them, and repair the running system.

4 Classification of Runtime Verification Tools

In this section we classify a number of RV tools with respect to the previously introduced taxonomy. The classification presented here is a revised and substantially extended version of the classification presented in our previous work [79]. To

our knowledge, this is the most comprehensive classification of the existing RV tools.

Tool selection. In our initial classification we considered a set of well documented and recently developed tools, which are still actively developed and maintained. We therefore focused on the tools that participated in the runtime verification competitions [23,81,139] taking place between 2014 and 2016 and the tools described in papers accepted at the RV-CuBES workshop [140], which took place in 2017. This led to an initial selection of 20 tools (14 chosen from the competition and 6 from the workshop) whose classification [79] was based on our analysis of the tools' relevant papers and documentation.

In order to validate our classification of the tools and further extend our tool selection, we have performed a field study, which aimed to collect the relevant information from the RV tools' authors directly. To that end, we have designed a questionnaire and disseminated it within the RV community.

Questionnaire. Our questionnaire[3] consists of seven sections each corresponding to a major concept in the taxonomy. Each section contains questions that address its sub-concepts. Since the version of the taxonomy at the moment of the questionnaire's inception may not be general (or specific) enough to classify all the tools, we allow the tool authors to provide a *custom* answer to each question in addition to the *fixed* answers (corresponding to the leaves in the taxonomy). The answers are not exclusive, i.e., the authors can select any subset of answers including the custom answer. Additionally, each question can be answered with *not applicable* if the tool does not fit that part of the taxonomy.

We consider the fixed answers *fitting*, while *custom* and *not applicable* answers as *non-fitting* answers. A tool's *fitness* is the fraction of its questionnaire answers that are fitting. In total, the questionnaire has 28 questions, where 26 are multi-choice-multi-answer and 2 free-form questions.

In addition to these questions, for each tool we have collected its name, references, link to its source code, as well as the email address of a contact person. Finally, in order to facilitate further improvements, we allowed the authors to provide their feedback on the taxonomy as part of a the last free-form question. We intend to keep the questionnaire open in order to be able to further revise our classification, add new tools, and receive feedback on the taxonomy.

Although the questionnaire is open to anyone, we have initially sent invitations to a targeted group of participants. The criteria for selecting the initial participants is the following:

- Any contact person for an RV tool (as specified on the tool's website)
- Any corresponding author of a paper featuring an RV tool
- Any author of a paper featuring an RV tool, with corresponding authors not specified
- Any program committee member of the RV conference in the last 3 years

The list of RV tools[4] used as a basis for deriving the list of participants has been initially created during the Dagstuhl seminar 17462 [95] and further populated by recursively considering all the tools referenced in the related work sections of tools' relevant papers.

Classification procedure. During the classification, we have followed an iterative procedure that alternated between the questionnaire's result analysis and the taxonomy refinement.

We have started from the existing classification [79] and unified it with the results of the questionnaire, ensuring that we can distinguish the data that came from the tool authors directly. Then, we have checked if each tool belongs to the general RV setting (as considered in this article) and we have eliminated those tools that do not. We have then analysed all the non-fitting answers. Given such an answer, for some tool and some part of the taxonomy, we read the provided tool references in order to understand how the tool fits that specific part of the taxonomy. If there is a fitting answer that the authors (potentially) misunderstood, we use it instead. Otherwise, we refine the taxonomy and check if any of the other tools need to be reclassified with respect to that part of the taxonomy.

Participating tools. The final list of names of 60 participating tools is shown in the leftmost column of Table 1 along with (where applicable) hyperlinks pointing to the tools' website or the source code. Other columns show tools' references, specification formalisms, and the description of the fragment of the specification formalism directly supported by each tool. The presented tools' references are sourced from the questionnaire participants and from additional information about the tools that can be found in the competition reports [23, 81, 139] or the RV-CuBES workshop proceedings [140].

Whenever possible, we denote a specification formalism with its usual acronym, as used in the corresponding tool's references. The tools that only use implicit specifications have the *not applicable* (na) symbol in this column.

The rightmost column briefly summarises the supported fragment of each specification formalism, as reported in the tool references or author's answers. Note that this concept is related to the monitorability/enforceability part of the taxonomy and applies only to the specification formalisms that have formal semantics. Typically, through the concept of supported fragments, a tool aims to characterise specifications for which its decision procedures guarantee some properties, or they are particularly efficient. Regarding this, some tools implement special provisions, like syntactic restrictions that characterise the supported fragment. Other tools analyse the input specification, possibly rewriting it into a supported form, and provide feedback to the user. Finally, some tools completely delegate

[3] Available at `https://forms.gle/mUiTZK3egSPr3Wsd9`

[4] Available at `https://goo.gl/Mmuhdd#gid=795731900`

Tool	References	Specification formalism	Supported fragment
Adapter	[47,48]	μHML	Safety fragment of μHML
Aerial	[26,27,32,33]	MTL, MDL	full
AgMon/EgMon	[108,109]	MTL	bounded future modalities
ARTiMon	[132,133]	ARTiMon language	bounded future modalities
AVA, BCT, and Radar	[13,118,124–126]	method pre- & post-conditions and FSM	full
BeepBeep 3	[91,92]	LTL-FO+, LoLA 1.0, QEA	full
Block-based atomicity checker (BBAC)	[165]	na	na
Breach	[66,67]	parametric STL	Online monitoring supports a limited parametric STL
Commit-node atomicity checker (CNAC)	[164]	na	na
Contract Larva	[75]	DEA	full
CPSDebug	[25]	STL	full
CRL	[44–46,127]	Chronicle language	full
DANA	[70]	Layered reference model	full
DecentMon	[36,55]	LTL	full
DejaVu	[96,98–100]	QTL	Closed formulas and finitely quantified variables in rigid predicates
detectEr	[10,11,50,84]	μHML	Safety and co-safety fragments of μHML
E-ACSL	[63,152]	E-ACSL	Summarised in the *implementation status* section of the tool's website
Eagle	[15,16,86]	Eagle	full
GREP	[142]	Timed automata	Timed languages described without using the clock difference
HYDRA	[134,135]	MTL, MDL	bounded future modalities
InterAspect	[151]	Regular expressions	full
JavaMOP	[107,119]	MOP plugins (LTL, FSM, ERE, CFG, SRS, ptCaReT)	full
JPaX	[102–105]	JPaX	full
jUnitRV	[61,62]	TDL	full
Larva	[56–58]	DATEs	full
LogFire	[94]	LogFire DSL	full
LogScope	[17,18,90]	LogScope	full
MarQ	[14,136,138]	QEA	full
Modbat	[8,9]	Scala assertions	na
MonPoly	[28–31]	MFOTL with aggregations	Safety fragment of MFOTL
Montre	[157,160,161]	TRE	full
MTLMapReduce (MTL-MR)	[38,39]	MTL	full
Mufin	[60]	Projection Automata	full
nfer	[110–112]	nfer	full
OpenJML	[53]	JML	Summarised in the *features* section of the tool's website
OptySim	[65,145]	LTL, Assertions	full
Orchids	[88,89,123]	Orchids specific	full
ParTraP	[42,51]	ParTraP	full
Proactive Libraries	[143,144]	Edit Automata with domain-specific extensions	full
Reelay	[158,159]	MTL, Regular expressions	Past MTL and Classical REs
RiTHM	[43,122]	LTL_3	LTL_3 without the NEXT operator
RML	[6,7]	RML	deterministic specifications
RMOR	[93]	RMOR language	full
R2U2	[121,141,149]	MTL, MLTL	bounded future modalities
RTC	[120]	na	na
RuleR	[20]	RuleR	full
RV-Monitor	[117]	MOP (see JavaMOP)	full
SOLOIST-ZOT	[37,40,41]	SOLOIST	propositional fragment
StateRover	[71]	UML statechart assertions	na
STePr		Scala-internal DSL	full
StreamLAB	[82]	RTLola	full
Striver	[87]	Striver	full
TemPsy-Check	[68,69]	TemPsy	full
THEMIS	[73,74]	LTL, Automata	full
TimeSquare	[59]	CCSL	full
TiPEX	[128,129]	Timed automata	full
TraceContract	[19]	TraceContract DSL	full
VALOUR	[12]	Valour Script & Rules	na

Table 1: Details of the classified tools

specification analysis to the user who needs to ensure that the tool is invoked with the appropriate specification. Typically, tools that support custom DSLs fall into this category and therefore we write that they support the full specification formalism. We do not classify the tools according to this criterion, but only report on the specific fragments, if they exist.

The authors of the MINT [163] tool have participated in the questionnaire. However, after reviewing their answers and the provided references, we have decided to exclude the tool from the final version of the classification. MINT is used for specification mining and iterative test-generation, which does not fit the scope of runtime verification, as defined in this article. Another indication that the tool is not suitable for classification is its fitness of 62%, which is significantly lower than the average tool fitness of 82%. Once the authors extend MINT to be applicable for online failure detection, which is stated as future work [163], we would be happy to reclassify the tool appropriately.

A similar concern applies in the case of the nfer tool, which we decided to keep in the classification. In addition to the specification mining, nfer can perform runtime verification tasks. This is confirmed by its fitness, which is 92%.

We have also eliminated the eAOP tool (fitness 69%). Indeed, an aspect-oriented programming framework can be used to perform RV tasks, but the monitors must be explicitly written. In our classification eAOP tool is subsumed by Adapter and detectEr tools, which are implemented using eAOP.

Finally, tools AVA, BCT, and Radar share many common characteristics and are often used together, hence they are also classified together.

Classification results. The classification of all the tools is given in Tables 3 and 4 with the acronyms described in Table 2. We leave a more detailed discussion of the classification for the next section. The classification also exists as a living document[5] and we welcome comments from the community, which we aim to incorporate both in the taxonomy and in the classification as our work continues.

The classification non-uniformly instantiates levels for different parts of the taxonomy. We omit parts of the taxonomy that are too abstract to be properly instantiated for the participating tools (e.g., system model and monitorability/enforceability), or if only one tool differs from all the others (e.g., THEMIS is the only tool that supports organising specifications in a decentralised manner).

The classification also refines the taxonomy. For instance, software instrumentation is refined based on its implementation (via AspectJ [113], Java reflection, eAOP [49], JTrek [52], native VM tracing, or the Xposed framework [2]). We also instantiate concrete implicit specifications and provide a more detailed description of the tools' decision procedures, whenever the tools' authors or references provide such information.

[5] Available at `https://goo.gl/Mmuhdd`

Besides the taxonomy-related values specified in Table 2, the cells in Tables 3 or 4 may contain values "all", or "none" indicating that the tool supports all, or none of the features defined by that part of the taxonomy. Value "na" states that this part of the taxonomy is not applicable to the tool, while "?" means that there is insufficient information about the tool to establish a definitive classification. The *Fitness* column shows the fitness of the tools based solely on the questionnaire answers. If there are multiple entries for the same tool, we report the tool's average fitness. We discuss how the final classification diverges from the authors' answers in Section 5. Tools whose authors participated in the questionnaire have the ✓ symbol in the *Author's input* column, while the others have the × symbol and the fitness measure does not apply to them.

Threats to validity. Whilst we believe that this is the most comprehensive classification of existing RV tools to date, there are three possible threats to its validity.

Firstly, the classified tools are sourced from academic tools developed within the runtime verification scientific community. This rules out commercial tools and tools from other closely-related communities, like databases, stream processing, and software engineering. Although some of these tools could fit the taxonomy quite well, the commercial tools are often badly documented, especially with respect to the concepts in the taxonomy. Furthermore, other communities do not share the same terminology with runtime verification, e.g., it is not usual to use a trace as an abstraction of a system.

Secondly, the classification focused on software monitoring with explicit specifications, which is a prominent research theme in the runtime verification community. Indeed, we have attempted to classify some tools that support only implicit specifications and identify themselves as RV tools (e.g. AdressSanitizer [150] or iflowTYPES.js [148]). Whilst such tools can be categorised in the taxonomy, their classification remains very coarse (with up to 30% fitness) and therefore such tools are not the focus of the classification. In general, this suggests that some areas of the taxonomy may require a refinement in the future, but also that these refinements will be orthogonal to the work presented here.

Lastly, the classification does not cover all known RV tools. Such tools are not classified due to authors' decision not to participate in the questionnaire. Still, within our defined scope, the coverage of tools is extensive.

5 Discussion

This section makes some observations about the taxonomy and the classification. We also discuss our classification process and provide details on particularly difficult classification cases. We believe that this section contributes to the provenance of the current state of the classification and can serve a list of initial discussion points for the further taxonomy refinements.

Column	Values
Specification	
implicit	ms = *memory safety*, at = *atomicity*, dmz = *division or modulo by zero*, ao = *arithmetic overflows*, bc = *large bit shifts*, dc = *illegal downcasts*, tc = *illegal typecasts*
data	p = *propositional*, s = *simple parametric*, c = *complex parametric*
output	sng(_) = *a single* _, seq(_) = *a sequence of* _, v = *verdict*, w = *witness*, r = *robustness*
logical time	tot = *total order*, par = *partial order*
physical time	di = *discrete*, de = *dense*, none = *no time*
modality	f = *future and current*, p = *past and current*, c = *current*
paradigm	d = *declarative*, o = *operational*
Monitor	
generation	e = *explicit*, i = *implicit*
execution	i = *interpreted*, d = *direct*
properties of the decision procedure	s = *soundness*, c = *completeness*, i = *impartiality*, a = *anticipation*
Deployment	
stage	on = *online*, off = *offline*
synchronisation	sync = *synchronous*, async = *asynchronous*
architecture	c = *centralised*, d = *decentralised*
placement	out = *outline*, in = *inline*,
instrumentation	sw = *software*, swAJ = *software with AspectJ*, swRF = *software with reflection*, swEA = *software with eAOP*, swJT = *software with JTrek*, swVM = *software with VM tracing*, swEX = *software with Xposed framework*
Interference	
Interference	in = *invasive*, ni = *non-invasive*
Reaction	
active	ex = *exception*, r = *recovery*, ro = *rollback*, en = *enforcement*
passive	so = *specification output*, e = *explanations*, st = *statistics*
Trace	
information	e = *events*, s = *states*
sampling	et = *event-triggered*, tt = *time-triggered*
evaluation	p = *points*, i = *intervals*
precision	p = *precise*, i = *imprecise*
model	f = *finite trace model*, i = *infinite trace model*
Application area	
Application area	pv = *property verification*, fp = *failure prevention & reaction*, td = *testing and debugging*, cq = *information collection & querying*
General	
	all = *all features supported*, none = *no features supported* na = *not applicable*, ? = *insufficient information*

Table 2: Abbreviations used in Tables 3, 4, and 5.

Tool	Specification							Monitor				Deployment					Interference	Reaction		Trace					Application area	Fitness	Author's input
		explicit						decision procedure																			
				time																							
	implicit	data	output	logical	physical	modality	paradigm	realisation	properties	generation	execution	stage	synchronisation	architecture	placement	instrumentation		active	passive	information	sampling	evaluation	precision	model			
Adapter	none	s	none	tot	none	na	d	automata-based	s	e	d	on	all	c	out	swEA	in	r	none	e	et	na	p	i	pv, fp	77%	✓
Aerial	none	p	seq(v)	tot	di	all	d	dynamic programming	s, c, i	i	i	all	sync	c	out	none	ni	none	so	e	et	p	p	i	pv	81%	✓
AgMon/EgMon	none	p	seq(v)	tot	di	all	d	dynamic programming	s, a	i	i	on	sync	c	out	none	ni	none	so	e	tt	p	p	i	pv	na	×
ARTiMon	none	s	seq(v)	tot	all	all	d	time function evaluation	s, i	i	d	all	sync	c	out	sw	in	?	so	s	et	i	p	i	pv, fp, td	81%	✓
AVA, BCT, and Radar	none	c	sng(w)	tot	none	na	all	automata-based	s, c	?	i	off	na	c	na	sw	in	na	e	e	et	p	p	i	td	73%	✓
BeepBeep 3	none	c	seq(v)	tot	none	all	all	stream transducers, automata-based	s, c, i	i	d	all	async	c	out	none	ni	none	so, st	e	et	p	p	i	pv, td, cq	87%	✓
BBAC	at	na	na	par	na	na	na	analytical	c	na	na	off	na	c	na	sw	in	na	e	e	et	na	p	f	pv	58%	✓
Breach	none	s	seq(v,w,r)	tot	de	f	d	analytical	?	i	i	all	sync	c	?	none	in	none	so, e	all	et	i	p	i	pv, td	na	×
CNAC	at	na	na	par	na	na	na	analytical	c	na	d	off	na	c	na	sw	in	na	e	e	et	na	p	f	pv	65%	✓
Contract Larva	none	s	sng(v)	tot	de	all	o	automata-based	s, c, i	i	d	on	sync	c	in	sw	in	r, ex, ro	na	all	et	p	p	f	pv, fp	92%	✓
CPSDebug	none	s	seq(w)	par	de	f	d	analytical	s, c	e	i	off	na	c	na	sw	in	na	e	all	tt	i	p	i	td	81%	✓
CRL	none	p	seq(v,w)	tot	di	f	d	duplicating automata	?	i	d	on	sync	c	out	none	in	none	so	e	et	p	p	i	pv, td	na	×
DANA	none	c	seq(v)	all	all	all	o	automata-based and rewrite-based	all	e	d	all	async	c	out	sw	ni	r, ex	all	e	all	p	p	all	all	96%	✓
DecentMon	none	p	sng(v)	tot	none	f	d	rewrite-based	s, c	i	i	off	na	d	na	none	ni	na	so, st	s	tt	p	p	i	pv, td, cq	81%	✓
DejaVu	none	s	seq(v)	tot	none	p	d	BDD operations	all	e	d	all	all	c	all	none	all	none	so	e	et	p	p	f	pv, td	81%	✓
detectEr	none	c	sng(v)	all	none	f	d	automata-based	all	e	d	on	async	d	out	swVM	all	none	so	e	et	p	p	all	pv	92%	✓
E-ACSL	ms, dmz, ao, dc	c	sng(v)	na	na	p	all	code rewriting	s, c	e	d	on	sync	c	in	sw	in	r, ex	none	s	et	p	na	i	pv, fp, td	81%	✓
Eagle	none	c	sng(v)	tot	all	all	d	rewrite-based	s, c, i	e	i	all	all	c	all	swAJ	in	ex	so	e	et	p	p	f	pv, fp, td	88%	✓
GREP	none	p	sng(v)	tot	de	f	o	automata-based	all	e	i	all	async	c	out	none	in	en	so	e	et	p	p	all	pv, fp	81%	✓
HYDRA	none	p	seq(v)	tot	di	all	d	rewrite-based	s, i	i	i	off	na	c	na	none	ni	na	so	e	et	p	p	i	pv	81%	✓
InterAspect	none	c	sng(w)	na	na	na	d	automata-based	s, c	e	i	on	sync	c	in	sw	in	none	e	e	et	na	p	f	pv, td	77%	✓
JavaMOP	none	s	sng(w)	tot	none	all	all	trace slicing, per plugin procedure	?	e	d	on	sync	c	in	swAJ	?	r	e	e	et	p	p	f	pv, td	na	×
JPaX	df, nd	p	sng(v)	tot	none	all	d	rewrite-based	s, c, i	all	d	all	async	c	out	swJT	in	ex	so	e	et	p	p	f	pv, td	92%	✓
jUnitRV	none	s	sng(v)	tot	none	f	d	SMT-based	?	e	d	on	sync	c	in	swRF	?	?	?	e	et	p	p	f	pv, td	na	×
Larva	none	c	sng(v)	tot	de	f	o	automata-based	?	e	d	on	sync	c	out	swAJ	in	ex, r	so	all	all	p	p	i	pv, cq	81%	✓
LogFire	none	c	sng(v, w)	tot	all	all	all	rewrite-based	all	i	d	all	all	c	all	none	in	none	so, e	e	et	p	p	f	pv, td, cq	85%	✓
LogScope	none	s	sng(v, w)	tot	all	f	all	automata-based	all	e	i	off	na	c	na	sw	ni	na	so, e	e	et	p	p	f	pv, td	85%	✓
MarQ	none	s	sng(v)	tot	all	f	o	automata-based	s, i, a	i	i	all	sync	c	out	swAJ	in	none	so	e	et	p	p	f	pv, td	88%	✓
Modbat	none	c	sng(v, w)	par	none	c	o	na	na	na	d	on	sync	c	in	sw	ni	none	all	e	na	p	p	na	td	58%	✓
MonPoly	none	s	seq(v, w)	tot	all	all	d	first-order queries	s, c, i	i	i	all	sync	c	out	none	ni	none	so	e	et	p	p	all	pv	79%	✓

Table 3: Classification of participating tools (part 1).

Tool	Specification							Monitor				Deployment					Interference	Reaction		Trace					Application area	Fitness	Author's input
		explicit						decision procedure																			
				time																							
	implicit	data	output	logical	physical	modality	paradigm	realisation	properties	generation	execution	stage	synchronisation	architecture	placement	instrumentation		active	passive	information	sampling	evaluation	precision	model			
Montre	none	p	seq(v, w)	tot	de	f	d	analytical and rewrite-based	s, c	i	d	all	sync	c	out	sw	ni	none	so, e	s	et	i	p	f	cq	88%	✓
MTL-MR	none	p	sng(v)	tot	di	all	d	analytical	s, c	e	i	off	na	d	na	none	ni	na	so	e	et	p	p	f	pv	85%	✓
Mufin	none	s	sng(v)	tot	none	f	o	automata-based (union-find)	?	i	d	on	sync	c	out	none	?	none	so	e	et	p	p	f	pv, td	na	×
nfer	none	s	seq(v,w)	par	de	all	d	analytical	s, c	e	all	all	async	c	all	sw	ni	none	so	all	et	i	p	f	pv, td, cq	92%	✓
OpenJML	ms, tc, dmz, bc	c	seq(v)	na	na	p	all	assertion checking	?	e	d	on	sync	c	in	sw	in	ex	all	all	et	p	na	?	pv, fp, td	na	×
OptySim	none	p	sng(v)	tot	none	f	d	automata-based	?	e	i	off	na	c	na	none	ni	none	so	all	et	p	p	i	pv, td	na	×
Orchids	none	c	sng(v, w, r)	tot	all	f	o	automata-based	s, i, a	e	all	all	async	c	out	sw	ni	ex	so	e	et	p	p	f	pv, fp	88%	✓
ParTraP	none	c	sng(w)	tot	de	all	d	analytical	s, c	na	i	off	na	c	na	sw	ni	na	e, st	e	et	p	p	f	pv, td	77%	✓
Proactive Libraries	none	c	seq(v)	tot	di	f	o	automata-based	s, c, i	e	d	on	sync	c	in	swEX	in	en	none	e	et	p	p	i	fp	88%	✓
Reelay	none	s	seq(v)	tot	all	p	d	dynamic programming	s, c	i	d	on	sync	c	out	sw	ni	none	so	all	et	all	p	f	pv, td	88%	✓
RiTHM	none	p	seq(v)	tot	none	f	o	time-triggered verification	?	e	d	on	async	c	in	sw	?	none	so	s	all	p	p	i	pv, td	na	×
RML	none	c	seq(v)	tot	none	na	d	rewrite-based	s, c, i	e	d	all	all	c	out	none	ni	none	so	e	et	p	p	all	pv, td	77%	✓
RMOR	none	p	sng(v)	tot	none	all	o	automata-based	all	e	i	on	sync	c	in	sw	in	ex	so	e	et	p	p	f	pv, fp, td	88%	✓
R2U2	none	p	seq(v)	tot	di	all	d	automata-based	?	e	i	on	async	c	out	none	?	none	so	e	et	p	p	i	pv, td	na	×
RTC	ms	na	na	na	na	na	na	?	?	i	d	on	sync	c	in	sw	?	r	?	?	et	p	p	na	fp	na	×
RuleR	none	c	seq(v)	tot	all	all	o	rewrite-based	all	e	i	all	all	c	all	swAJ	in	ex	so	e	et	p	p	f	all	88%	✓
RV-Monitor	none	s	sng(w)	tot	di	all	all	(see JavaMOP)	?	i	d	all	sync	c	all	sw	?	r	e	e	et	p	p	f	pv, fp, td	na	×
SOLOIST-ZOT	none	p	sng(v)	tot	di	all	d	SMT-based	s, c, i	e	i	off	na	c	na	none	ni	na	so	e	et	p	p	i	pv	85%	✓
StateRover	none	c	sng(v)	tot	di	f	o	automata-based	s	na	d	all	all	d	all	sw	ni	none	so	e	all	p	p	f	pv, td, cq	88%	✓
STePr	none	s	seq(v)	tot	di	all	o	?	?	i	d	on	?	c	out	none	?	none	so	e	et	p	p	?	pv, td	na	×
StreamLAB	none	s	seq(v)	tot	de	all	o	Stream-based	all	e	i	all	async	c	out	none	ni	none	so, st	e	all	na	na	f	pv, td, cq	69%	✓
Striver	none	c	seq(v)	tot	de	all	d	automata-based	s, i, a	e	i	on	async	c	out	none	ni	none	so	e	et	p	p	all	pv, td, cq	88%	✓
TemPsy-Check	none	p	sng(v)	tot	di	f	d	OCL constraint solver	s, c	i	all	off	na	c	na	none	ni	na	so	e	et	p	p	f	pv, cq	77%	✓
THEMIS	none	p	seq(v)	tot	di	all	all	automata-based	s, i	e	i	all	all	d	all	sw	ni	ex	so, st	s	tt	p	p	f	pv, td, cq	96%	✓
TimeSquare	none	p	seq(v)	par	all	all	d	CCSL evaluation	?	i	d	all	sync	c	out	none	ni	none	so	e	all	all	p	all	pv, td	na	×
TiPEX	none	p	seq(v)	tot	de	f	o	automata-based	s	i	i	on	async	c	out	none	in	en	so	e	et	p	p	f	fp	73%	✓
TraceContract	none	c	sng(w)	tot	di	all	all	rewrite-based	s, c, i	i	d	all	all	c	all	none	in	r, ex	all	e	et	p	p	f	all	92%	✓
VALOUR	none	s	sng(v)	tot	di	all	o	automata-based	?	i	d	on	all	c	in	swAJ	?	none	all	e	all	p	p	f	pv, td, cq	na	×

Table 4: Classification of participating tools (part 2).

5.1 General Remarks and Underdeveloped Taxonomy Parts

Here we discuss some general observation on the sample of tools that were classified according to our taxonomy. We also discuss details and differences between the tools that are not properly captured by the taxonomy due to its generality.

Specification. The majority of the classified tools use explicit specifications. Among them, the majority considers totally-ordered logical time. There is an even split between propositional and parametric tools and between the tools with different approaches to physical time.

Among the tools supporting implicit specifications, only BBAC, CNAC, and RTC tools do not additionally support some form of explicit specification. Such tools are hard to classify according to the concepts related to the explicit specification. Tools that support both types of specification are E-ACSL, JPaX, and OpenJML.

Regarding the data support in explicit specifications, there is an (almost) uniform presence of the tools that support propositional, simple parametric, and complex parametric specifications. Note that in our classification we assume that the support for complex objects as parameters subsumes the support for simple (primitive) parameters, which in turn subsumes the support for propositions. We normalised questionnaire answers with respect to this assumption.

Most of the tools use discrete time domain in their specification formalism, with 10 tools using both discrete and dense time domains. Some tools that use discrete time (e.g., Reelay) interpret the received observations as if (implicitly) separated by a unit of physical time. Others (e.g., Aerial) use explicit timestamps, assumed available in the input. Unfortunately, our taxonomy does not distinguish between these two ways of interpreting physical time. In general, the concept of physical time should be extended beyond only considering how the tool's specification formalism defines the time domain. It should additionally describe how the tool handles time in practice. The current taxonomy is imprecise as, for instance, we assume that tools working with the dense time domains use an appropriate dense time approximation in their implementations.

Tools DANA, Eagle, LogScope, MarQ, Orchids, and RuleR use a time domain based on the data they read from the trace. We therefore decided to classify them as working with both discrete and dense time domains.

For the physical time, which is explicitly available in the form of timestamps, it is worth noting *when* it was measured. Such a concern is widely known in stream processing, which distinguishes between various moments when the input of a stream processor is tagged with physical time. In the RV setting, if an observation is tagged at the moment of its occurrence within the monitored system, it is tagged with its *creation time*. This corresponds to *event time* in stream processing terminology [5]. If an observation is tagged when it is sent to the monitor, it is tagged with its *emission time*. Outline RV tools that do not provide their own instrumentation often assume that their input contains either the creation or emission times. The RV tools that do not receive timestamped observations, tag them either with their *ingestion time* (i.e., the time an observation is received), or with their *processing time* (i.e., the time when an observation is first used in the monitor's internal computation). Our taxonomy is not refined to include this distinction, although some classified tools differ with respect to this concept.

The distinction between operational and declarative specification languages results in two sets of tools of roughly the same size. A total of 10 tools supports both paradigms.

Monitor. Some parts of the taxonomy were fairly straightforward to complete (e.g., deployment), whereas others were more controversial. The most discussed part of the taxonomy was the monitor concept as the term "monitor" is highly overloaded in the RV community and many approaches do not explicitly define the notion of a monitor. In the end, we decided to split monitor generation from its execution, as there is not necessarily a close link between the two concepts.

Although each tool uses custom realisations of its decision procedures, many of the tool authors opted for the (more generic) fixed answer (e.g., analytical, automata-based, or rewrite-based) in the questionnaire. This shows that despite the versatility of the decision procedures, we managed to achieve a good high-level classification.

Regarding the properties of the decision procedures, we have to stress that the values reported in the classification are those provided by the authors. Based on our assessment of the questionnaire answers, we can conclude that, in most cases, the authors did not consider the properties as defined in this article, but rather as they are defined in the respective tool references. Nevertheless, we decided to include this information in the classification.

Deployment. The majority of tools are online – it is perhaps worth observing that RV-Monitor added an offline interface for the competition. In our previous classification [79] only one tool was purely offline, while classification in this article contains 13 such tools.

The exclusively offline tools have *na* in the synchronisation column. A larger fraction of the rest of the (online) tools are synchronous rather than asynchronous.

Unlike in our previous classification [79], this classification contains five tools with a decentralised architecture. THEMIS tool is the only tool that additionally supports a decentralised specification formalism. Although initially declared as decentralised by the tool authors, StreamLAB and Proactive Libraries are classified as centralised, since there is not enough evidence to support the tools' decentralised nature.

Regarding the monitoring placement, outline tools are more commonly developed. Outline tools are typically domain-independent and designed as general-purpose tools. This is confirmed by the fact that all tools that do not provide instrumentation (which is problem-specific) are outline. Note that, according to our taxonomy offline tools do not have a placement and hence are classified as *na* with respect to this criterion. However, our definition of placement allows us to treat offline tools as outline, since they execute in a separate memory space from the monitored system.

Interference. The concept of interference is one of the most underdeveloped parts of the taxonomy. Unfortunately, the questionnaire answers were mostly fixed, hence not helpful in refining the taxonomy. The only conclusive trend is that offline tools that do not provide instrumentation are non-invasive.

Reaction. Almost all the tools provide some form of output, which we classify as a passive reaction. In our classification we see that the offline tools do not provide active reaction, which means that there is a relation between these two major concepts in our taxonomy. Based on our classification GREP, Proactive Libraries, and TiPEX are the only tools that provide full specification enforcement. Besides active reaction, some of these tools provide a passive output as well. For instance GREP outputs additional feedback on whether the outcome of the enforcement was successful.

Other RV tools that allow for some form of active reaction either stop the system with an exception or call some of the recovery routines. Contract Larva is the only tool that allows rolling back of the system to a last correct state. The authors of Eagle, JPaX, Larva, Orchids, RMOR, and RuleR provided similar custom answers in the questionnaire. Namely, they allow for custom user code to be executed upon specification violation. We decided to classify them as raising exceptions and we note that our taxonomy is not precise enough to distinguish whether the exceptions are handled with pre-defined or user-defined procedures.

The RV tools that exclusively react in a passive way usually provide the specification output as their output. BeepBeep 3, DecentMon, ParTraP, StreamLAB, and THEMIS provide statistics, while Breach, CPSDebug, InterAspect, JavaMOP, LogFire, LogScope, Montre, ParTraP, and RV-Monitor provide some form of explanations. Finally, DANA, Modbat, OpenJML, TraceContract, and VALOUR provide output that covers all possibilities enumerated in our taxonomy. We note that, the taxonomy is not exhaustive in this respect, since some RV tools provide other kinds of output. For instance, MonPoly checks if its input specification belongs to the monitorable fragment and if not provides feedback to the user. Such monitoring output is not covered by our taxonomy.

Trace. Another area that was difficult for the classification effort was the relation between the trace model and the observed trace. It would be wrong to conflate the two, however often these concepts overlap. On the other hand, many tools do not formally express the notion of a trace, which makes this part of the classification significantly harder.

Almost all the tools consider events as observations and use event-triggered sampling. ARTiMon, DecentMon, E-ACSL, Montre, RiTHM, and THEMIS tools only consider states. Some event-triggered tools support timers (e.g., Larva), hence we also consider them to use time-triggered sampling.

All tools monitor precise traces and a significant majority evaluates specifications over observations made at individual time points. Tools ARTiMon, Breach, CPSDebug, Montre, nfer, and Reelay evaluate specifications over observations coming from time intervals, with Reelay performing the evaluation over both time points and intervals in the trace.

The trace part of our taxonomy focuses on the characteristics of a single trace. This excludes research efforts from the runtime verification community that deal with hyperproperties, i.e., properties of sets of traces. This may explain why no RV tools for hyperproperties such as RVHyper [83] participated in our questionnaire. On the other hand, the same characteristics of traces (e.g., denoting which information they carry and how they are evaluated) apply to sets of traces. Nevertheless, we leave the precise characterisation of RV tools for hyperproperties within our taxonomy as future work.

Application area. Many of the tools were not developed with a concrete application area in mind, making this part of the taxonomy less relevant. However, in cases where an application exists it is significant. For example, R2U2 is designed to monitor unmanned aerial vehicles and this is heavily reflected in the tool's design.

Although majority of questionnaire answers chose *property verification* (which is a part of the *analysis* concept), this part on the taxonomy is very interesting when refined, especially in certain cases when the application area significantly influences the tool design.

5.2 Multiple classification entries

During the process of classification, some tools were classified multiple times. Specifically, Aerial, ARTiMon, BeepBeep 3, DANA, detectEr, Larva, LogFire, MarQ, MonPoly, and TemPsy-Check tools have duplicate entries in the classification, coming either from our previous classification [79] or from multiple answers provided by the tool authors in the questionnaire. The classification entries provided by us are denoted with the × symbol in Table 5 in the *Author's input* column. Otherwise, tool author's entries are denoted with the ✓ symbol. All entries in Table 5 are shown in their genuine

Tool	Specification							Monitor				Deployment					Interference	Reaction		Trace					Application area	Author's input
		explicit						decision procedure																		
				time																						
	implicit	data	output	logical	physical	modality	paradigm	realisation	properties	generation	execution	stage	synchronisation	architecture	placement	instrumentation		active	passive	information	sampling	evaluation	precision	model		
Aerial	none	p	seq(v)	tot	di	all	d	dynamic programming	?	i	i	on	none	c	out	none	?	none	so	e	et	p	p	i	?	✓
Aerial	none	p	seq(v)	tot	di	all	d	dynamic programming	s, c, i	i	i	all	na	c	out	na	ni	na	so	e	et	p	p	i	pv	✓
ARTiMon	none	s	seq(v)	tot	all	all	d	?	?	i	i	on	none	c	out	none	?	none	so	e	et	i	p	i	?	×
ARTiMon	none	s	seq(v)	tot	all	all	d	**Time functions (...)**	s, i	i	d	all	sync	c	out	sw	in	**Depends on the usage**	**Depends on the usage**	**Timed states**	et	i	p	i	pv, fp, td	✓
BeepBeep 3	none	c	seq(v)	tot	none	f	all	stream-processing	?	i	d	on	all	c	out	sw	?	none	so	e	et	p	p	i	?	×
BeepBeep 3	none	all	seq(v)	tot	di	all	all	transducers on streams	s, c, i	all	i	all	async	c	out	na	ni	na	so	e	et	p	p	i	pv, td, cq	✓
BeepBeep 3	none	c	seq(v)	tot	di	all	all	automata-based	s, c, i	all	all	all	async	c	out	na	ni	na	so, st	e	et	p	p	i	pv, td, cq	✓
DANA	none	p	seq(v)	tot	de	all	o	?	?	i	d	on	sync	c	?	?	?	none	so	e	et	p	p	f	?	×
DANA	none	all	seq(v)	all	**data**	all	o	automata- and rewrite-based	all	all	all	all	async	c	out	sw	ni	r, ex	all	all	all	p	p	all	all	✓
detectEr	none	s	sng(v)	par	none	f	d	dynamic programming	?	e	i	on	all	c	in	sw	?	none	so	e	et	p	p	i	?	×
detectEr	none	c	sng(v)	all	di	f	d	automata-based	s, i, a, **(partial) c**	e	d	on	async	all	out	swVM	all	na	so	e	et	p	p	all	pv	✓
Larva	none	s	sng(v)	tot	di	f	o	automata-based	?	e	d	on	all	c	all	sw	?	r	so	e	et	p	p	f	?	×
Larva	none	all	sng(v)	tot	de	na	o	automata-based	**Depends on the usage**	na	d	on	sync	c	out	swAJ	in	**user specified**	so	all	all	p	p	i	pv, cq	✓
LogFire	none	s	sng(w)	tot	none	all	o	RETE	?	i	d	all	sync	c	out	none	?	none	so	e	et	p	p	f	?	×
LogFire	none	all	sng(v, w)	tot	**data**	all	all	rewrite-based	all	i	d	all	all	c	all	na	in	na	so, e	e	et	p	p	f	pv, td, cq	✓
MarQ	none	s	sng(v)	tot	di	f	o	automata-based	?	i	d	all	sync	c	all	sw	?	none	so	e	et	p	p	f	?	✓
MarQ	none	s	sng(v)	tot	**data**	f	o	automata-based	s, i, a	i	i	all	sync	c	out	swAJ	in	na	so	e	et	p	p	f	pv, td	✓
MonPoly	none	s	seq(v)	tot	di	all	d	first-order queries	?	i	i	on	none	c	out	none	?	none	so	e	et	p	p	all	?	×
MonPoly	none	p, s	seq(v, w)	tot	di	all	d	analytical	i, s, c	na	i	all	async	c	out	na	ni	na	so	e	et	p	na	i	pv	×
MonPoly	none	s	seq(v)	tot	di	all	d	analytical	s, c	na	i	all	async	c	out	na	na	na	so	e	na	p	p	i	pv, td	✓
TemPsy-Check	none	p	sng(v)	tot	di	all	d	OCL constraint solver	?	i	i	off	na	c	out	none	?	none	so	e	et	p	p	f	?	×
TemPsy-Check	none	p	sng(v)	tot	di	f	d	analytical	s, c	i	all	off	na	c	na	na	na	na	all	e	et	p	p	f	pv, cq	✓

Table 5: Duplicate tool entries encountered during the classification with the custom answers shown in **bold**.

form, with custom answers shortened for space reasons and denoted in bold. The classification discrepancies in cases of Aerial and MarQ tools are particularly interesting and may hint at our own inconsistent treatment of the taxonomy, since we are also the tools' authors.

All the taxonomy concepts where the multiple classification entries agree were adopted into the final classifications. The decision procedure *properties*, *interference*, and *application area* concepts are not present in the previous classification [79], hence we simply adopt the new values instead of the insufficient information (?) symbol for all the tools. In the case of the non-highlighted concepts where disagreement exists we adopted the input from the authors.

In some cases the changes in the final classification are due to normalisation of the classification values explained previously. For instance, value c in the *data* column subsumes the other values and essentially means *all*. Similarly, value s subsumes p, hence we write p,s instead. The custom answer *data* in the *physical time* column is replaced with value *all*. In the *architecture* column we changed all *all* values into *d* values, since decentralised tools can be trivially run in a centralised fashion. If a tool does not provide instrumentation we classify it with value *none*, rather than *na*. In the *active reaction* column, offline tools are classified with value *na*, since they are not able to perform any active reaction. On the other hand, online tools that do not provide any active reaction are classified with a value *none*. The rest of the cases we highlighted in grey and we discuss them individually.

The Aerial tool is classified twice by us and the classifications are (mostly) consistent. The tool is designed to be online, but can be used for offline monitoring as well, which explains the difference in the *stage* column. After reviewing Aerial's code, we classified it as a synchronous tool. Both of the original answers ignore synchronisation due to the misconception that a tool when run as a black box can consume multiple events without explicitly halting to produce its output. However, this is facilitated by operating system's buffers, which have a limited size.

ARTiMon obtains information about timed states from its input trace, hence we classify its trace information to be *states*. The vague custom answers in the reaction part of the taxonomy led us to review ARTiMon's documentation. However, we were unable to find any details about its active reaction capabilities, nor about configurable passive reaction.

Regarding BeepBeep 3, the sources of the three classification entries are two questionnaire answers by the author and our previous classification. The main difference between the two answers from the author is the decision procedure, so we decided to keep the second (more general) entry in our main classification and list the two decision procedures explicitly. We opted to write *none* for its physical time support, since this is the trait of the specification language and BeepBeep 3 does not support metric specifications. The generation and execution parts of the taxonomy have a very general description, hence understandably they are hard to discern. After analysing the tool more closely we decided to classified as implicitly generated and directly executed.

After considering DANA's documentation, we concluded that it focuses on events as the source of the trace information (rather than on both events and states). The tool relies on stereotyped UML state machines to specify allowed sequences of events, which are obtained from messages generated by the system through a mapping defined in the interface and event definition models [70]. After studying the tool more closely we decided to classify it as explicitly generated and directly executed.

We treat detectEr's partial completeness as completeness, since the definition of completeness changes for each tool. Still, we document its specification formalism's supported fragment in Table 1.

Larva supports dynamic automata with timers and events (DATEs) as its specification formalism. Since DATEs are forward-reading automata, we classify Larva to support future modalities. As for the properties of Larva's decision procedure, we did not have enough information to obtain a definitive classification. We also added the support for *exceptions* in Larva's active reaction classification, which is consistent with our classification of other tools where authors provided a similar custom answer. After studying the tool more closely we decided to classify it as explicitly generated.

Regarding MonPoly, we have classified it to additionally provide witnesses as a part of its specification output, since for each verdict it outputs bindings of free variables that violate the specification. After a closer inspection of the tool, we realised that MonPoly supports both discrete and dense physical time. Interestingly, the classification entries from multiple authors all claimed discrete time support only. We also classified MonPoly as an implicitly generated monitor, since it does not explicitly generate code for its input specifications. MonPoly processes events one by one as they arrive (hence it is event-triggered) and it can be invoked (by means of a flag) to monitor based on either MFOTL's finite or infinite trace semantics. We also adopt a more refined description of its decision procedure.

Finally, the TemPsy-Check's companion tool provides richer passive reaction (e.g., explanations), but not TemPsy-Check itself. Hence, we opt for a more conservative classification. Like in the case of MonPoly, we use a more descriptive variant of the decision procedure.

5.3 Underpopulated Parts of the Taxonomy

The classification unveils parts of the taxonomy that are not populated by any tools. We discuss the main ones here and hypothesise as to why this might be the case.

Decentralised architecture. Decentralisation seems to be an area that has not received much attention. It may be due to its inherent complexity. There may also be inter-dependencies with the monitoring setting (e.g. the language of interest) that make such an approach more complex. Furthermore, the selection of the RV tools based on the participation in the competitions might have contributed to this topic being underrepresented: the competitions did not focus on the distributed setting.

Monitoring states. Only a small number of the tools in our classification monitor states of a program directly. This could be a result of the popularity of event-oriented specification formalisms. It could also be caused by the bias of the RV community (and thus our classification) towards tools with explicit specifications, while many of the tools that support implicit specifications work with state information. We have remarked this in the paragraph on the threats to validity in Section 4. Nevertheless, our classification provides an interesting insight: the commonly stated dichotomy between observing events or states is not reflected in practice. Furthermore, the distinction between states and events is not always clear: it is always possible to encode state as a pair of (start and end) events, while some inline tools allow specifications to directly query the current runtime state of the monitored system.

Richer reactions. Most tools only provide passive reactions and the active reactions provided were fairly weak. It would be interesting to see more work in the areas of enforcement, recovery, and explanations for declarative specifications.

Hardware instrumentation. All the classified tools either provide software instrumentation or no instrumentation at all. We hypothesise that R2U2 has some hardware instrumentation capabilities due to its application area. However, we have not received any direct input from the tool's authors. Besides the influence of the application area, hardware instrumentation may arise from the monitor implementation technologies. For instance, in approaches where monitors are compiled directly to configurable hardware components.

Monitoring imprecise traces. None of the classified tools support imprecision in their input traces. Approaches dealing with some types of imprecision exist. For instance, monitoring traces with imprecise timestamps [35], or traces with incomplete events or inconsistencies in event sources [34], or even purposefully omitting events when sampling [155]. Authors of the tools that support these ideas have not participated in our questionnaire.

5.4 Relation to the Previous Taxonomy

We briefly compare our taxonomy to the previous most complete taxonomy for runtime verification [64]. The context of this taxonomy is slightly different as their focus was software-fault monitoring. We have chosen to focus more on issues related to the monitoring of *explicit specifications* and include fewer operational issues. Delgado et al. identify four top-level concepts: *Specification*, *Monitor*, *Event-Handler*, and *Operational Issues*. Below we summarise the most significant differences in each area.

Specification. In the previous taxonomy the focus is more on the kind of property being captured (e.g. safety) and the abstraction at which the property is captured (e.g. whether it directly refers to implementation details). There is little discussion on handling of data or modalities (although one concept is *language type* which may be algebra, automata, logic, or HL/VHL). They also consider which parts of a program are (or can be) instrumented as part of the specification.

Monitor. Here, Delgado et al. again focus on the instrumentation, which this article does not consider in depth. We rather tend to draw a clear line between instrumentation and monitoring. They differentiate whether instrumentation is manual or automatic. The key observation here is that they view *placement* slightly differently, as they classify monitoring occurring using different resources (e.g. running in a different process) as *offline*. We refer to [24] for a discussion on the recent alternatives when considering instrumentation.

Event-Handler. This concept corresponds to our concept of *reaction* and its sub-concepts are subsumed by ours.

Operational issues. This is a concept that we have not considered in our taxonomy. It focuses on different *source program types*, i.e., the types of tools that can work in limited operational environments. For example, it is relevant whether an RV tool only works with a monitored system written in Java, or with a particular *dependency* available (e.g., a specific piece of hardware). This concept emphasises the overall *maturity* and *usability* of an RV tool.

While we think that our taxonomy would benefit from such a concept, the classification of RV tools would become significantly more difficult. In practice, we found that many RV tools are actively developed and data relevant for this concept may quickly become outdated.

Application areas. The chosen categories in the application areas part of the taxonomy are somewhat subjective and should be thought as examples. A rigorous classification of application areas for runtime verification is out of the scope of this article. We refer to [147] for a recent discussion on

the future challenges of runtime verification in various application areas categorised as distributed systems, hybrid and embedded systems, hardware, security and privacy, transactional information systems, contracts and policies, and huge, reliable or approximated domains.

6 Conclusion and Future Work

We have introduced a taxonomy for classifying runtime verification tools and used it to classify 60 tools. To our knowledge, this is the most comprehensive classification of existing runtime verification tools.

We believe that the proposed taxonomy and the classification activity carried out in this article are important for a number of reasons. Firstly, the taxonomy fixes shared terminology and dimensions for discussing RV tools – it is very important that the community has a shared language for what it does. Secondly, the classification exercise gives an overview of comparable tools, making it more straightforward to identify the tools against which new contributions should be compared. Additionally, the taxonomy can help shape evaluation and benchmark activities in general, in particular the design of competitions. Finally, we believe this kind of activity can identify interesting directions for future research, in particular in underpopulated areas of the taxonomy.

We consider our work ongoing and plan to continue refining the proposed taxonomy and extending the tool classification with new tools. We are also working on a visualisation tool that would provide an easier way to interactively browse, update, and extend the taxonomy and the classification.

Acknowledgement. The authors warmly thank Martin Leucker for the early discussions on the taxonomy and its mind map representation. This article is based on work from COST Action ARVI IC1402 [4], supported by COST (European Cooperation in Science and Technology). In particular, the taxonomy and classification benefited from discussions within working groups one and two of this action. We would also like to acknowledge input from participants of Dagstuhl seminar 17462 [95]. We thank anonymous RV and STTT reviewers for their input that helped us to improve the presentation of this work and to refine the taxonomy. Finally, we would like to thank all the tool authors who contributed to the tool classification. The research is partially supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162), the US Air Force grant “Monitoring at Any Cost” (FA9550-17-1-0306), and by H2020-ECSEL grants CPS4EU 2018-IA call - Grant Agreement number 826276.

References

1. IC1402 Runtime Verification beyond Monitoring (ARVI). `https://www.cost-arvi.eu/`
2. Xposed. URL `https://repo.xposed.info/`
3. Ahishakiye, F., Jaksic, S., Lange, F.D., Schmitz, M., Stolz, V., Thoma, D.: Non-intrusive MC/DC measurement based on traces. In: D. Méry, S. Qin (eds.) 2019 International Symposium on Theoretical Aspects of Software Engineering, TASE 2019, Guilin, China, July 29-31, 2019, pp. 86–92. IEEE (2019). DOI 10.1109/TASE.2019.00-15
4. Ahrendt, W., Artho, C., Colombo, C., Falcone, Y., Krstić, S., Leucker, M., Lorber, F., Lourenço, J., Mariani, L., Sánchez, C., Schneider, G., Stolz, V.: COST action IC1402 ARVI: Runtime verification beyond monitoring – activity report of working group 1. CoRR **abs/1902.03776** (2019)
5. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB **8**(12), 1792–1803 (2015)
6. Ancona, D., Ferrando, A., Mascardi, V.: Comparing trace expressions and linear temporal logic for runtime verification. In: E. Ábrahám, M.M. Bonsangue, E.B. Johnsen (eds.) Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, *LNCS*, vol. 9660, pp. 47–64. Springer (2016)
7. Ancona, D., Ferrando, A., Mascardi, V.: Parametric runtime verification of multiagent systems. In: K. Larson, M. Winikoff, S. Das, E.H. Durfee (eds.) Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017, pp. 1457–1459. ACM (2017)
8. Artho, C., Seidl, M., Gros, Q., Choi, E., Kitamura, T., Mori, A., Ramler, R., Yamagata, Y.: Model-based testing of stateful apis with modbat. In: M.B. Cohen, L. Grunske, M. Whalen (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, pp. 858–863. IEEE Computer Society (2015)
9. Artho, C.V., Biere, A., Hagiya, M., Platon, E., Seidl, M., Tanabe, Y., Yamamoto, M.: Modbat: A model-based API tester for event-driven systems. In: V. Bertacco, A. Legay (eds.) Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings, *LNCS*, vol. 8244, pp. 112–128. Springer (2013)
10. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Y. Falcone, C. Sánchez (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, *LNCS*, vol. 10012, pp. 473–481. Springer (2016)
11. Attard, D.P., Francalanza, A.: Trace partitioning and local monitoring for asynchronous components. In: A. Cimatti, M. Sirjani (eds.) Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings, *LNCS*, vol. 10469, pp. 219–235. Springer (2017)
12. Azzopardi, S., Colombo, C., Ebejer, J.P., Mallia, E., Pace, G.: Runtime verification using VALOUR. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 10–18. EasyChair (2017)
13. Babenko, A., Mariani, L., Pastore, F.: AVA: automated interpretation of dynamically detected anomalies. In: G. Rothermel, L.K. Dillon (eds.) Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009, pp. 237–248. ACM (2009)
14. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: D. Giannakopoulou, D. Méry (eds.) FM 2012, *LNCS*, vol. 7436, pp. 68–84. Springer (2012)

15. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA. IEEE Computer Society (2004)
16. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: B. Steffen, G. Levi (eds.) Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings, *LNCS*, vol. 2937, pp. 44–57. Springer (2004)
17. Barringer, H., Groce, A., Havelund, K., Smith, M.H.: An entry point for formal methods: Specification and analysis of event logs. In: M.L. Bujorianu, M. Fisher (eds.) Proceedings FM-09 Workshop on Formal Methods for Aerospace, FMA 2009, Eindhoven, The Netherlands, 3rd November 2009., *EPTCS*, vol. 20, pp. 16–21 (2009)
18. Barringer, H., Groce, A., Havelund, K., Smith, M.H.: Formal analysis of log files. JACIC **7**(11), 365–390 (2010)
19. Barringer, H., Havelund, K.: Tracecontract: A scala DSL for trace analysis. In: M.J. Butler, W. Schulte (eds.) FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings, *LNCS*, vol. 6664, pp. 57–72. Springer (2011)
20. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from eagle to ruler. J. Log. Comput. **20**(3), 675–706 (2010)
21. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: B. Bonakdarpour, S.A. Smolka (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, *LNCS*, vol. 8734, pp. 1–9. Springer (2014)
22. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, *LNCS*, vol. 10457. Springer (2018)
23. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. STTT pp. 1–40 (2017)
24. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: E. Bartocci, Y. Falcone (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, *LNCS*, vol. 10457, pp. 1–33. Springer (2018)
25. Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Nickovic, D.: Automatic failure explanation in CPS models. CoRR **abs/1903.12468** (2019). URL `http://arxiv.org/abs/1903.12468`
26. Basin, D., Bhatt, B.N., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. Formal Methods in System Design (2019)
27. Basin, D., Bhatt, B.N., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: A. Legay, T. Margaria (eds.) TACAS 2017, *LNCS*, vol. 10206, pp. 94–112. Springer (2017)
28. Basin, D., Harvan, M., Klaedtke, F., Zalinescu, E.: MONPOLY: monitoring usage-control policies. In: S. Khurshid, K. Sen (eds.) RV 2011, *LNCS*, vol. 7186, pp. 360–364. Springer (2011)
29. Basin, D., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design **46**(3), 262–285 (2015)
30. Basin, D., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
31. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 19–28. EasyChair (2017)
32. Basin, D., Krstić, S., Traytel, D.: AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 29–36. EasyChair (2017)
33. Basin, D., Krstić, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: S.K. Lahiri, G. Reger (eds.) RV 2017, *LNCS*, vol. 10548, pp. 85–102. Springer (2017)
34. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring compliance policies over incomplete and disagreeing logs. In: S. Qadeer, S. Tasiran (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, *LNCS*, vol. 7687, pp. 151–167. Springer (2012)
35. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: On real-time monitoring with imprecise timestamps. In: B. Bonakdarpour, S.A. Smolka (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, *LNCS*, vol. 8734, pp. 193–198. Springer (2014)
36. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design **48**(1-2), 46–93 (2016)
37. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., San Pietro, P.: SMT-based checking of SOLOIST over sparse traces. In: S. Gnesi, A. Rensink (eds.) Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, *LNCS*, vol. 8411, pp. 276–290. Springer (2014)
38. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., San Pietro, P.: Efficient large-scale trace checking using mapreduce. In: L.K. Dillon, W. Visser, L. Williams (eds.) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pp. 888–898. ACM (2016)
39. Bianculli, D., Ghezzi, C., Krstić, S.: Trace checking of metric temporal logic with aggregating modalities using mapreduce. In: D. Giannakopoulou, G. Salaün (eds.) Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings, *LNCS*, vol. 8702, pp. 144–158. Springer (2014)
40. Bianculli, D., Ghezzi, C., Krstić, S., San Pietro, P.: Offline trace checking of quantitative properties of service-based applications. In: 7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014, pp. 9–16. IEEE Computer Society (2014)
41. Bianculli, D., Ghezzi, C., San Pietro, P.: The tale of SOLOIST: A specification language for service compositions interactions. In: C.S. Pasareanu, G. Salaün (eds.) Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers, *LNCS*, vol. 7684, pp. 55–72. Springer (2012)
42. Blein, Y., Ledru, Y., du Bousquet, L., Groz, R.: Extending specification patterns for verification of parametric traces. In: S. Gnesi, N. Plat, P. Spoletini, P. Pelliccione (eds.) Proceedings of the 6th Conference on Formal Methods in Software Engineering, FormaliSE 2018, collocated with ICSE 2018, Gothenburg, Sweden, June 2, 2018, pp. 10–19. ACM (2018)
43. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Time-triggered runtime verification. Formal Methods in System Design **43**(1), 29–60 (2013)
44. Carle, P., Choppy, C., Kervarc, R.: Behaviour recognition using chronicles. In: Z. Duan, C.L. Ong (eds.) 5th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2011, Xi'an, China, 29-31 August 2011, pp. 100–107. IEEE Computer Society (2011)
45. Carle, P., Choppy, C., Kervarc, R., Piel, A.: Handling breakdowns in unmanned aircraft systems. In: FM 2012: Formal Methods - 18th International Symposium - Doctoral Symposium, Paris, France, August 27-31, 2012. Proceedings (2012)

46. Carle, P., Choppy, C., Kervarc, R., Piel, A.: Safety of unmanned aircraft systems facing multiple breakdowns. In: C. Choppy, J. Sun (eds.) 1st French Singaporean Workshop on Formal Methods and Applications, FSFMA 2013, July 15-16, 2013, Singapore, *OASICS*, vol. 31, pp. 86–91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
47. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: E. Bartocci, R. Majumdar (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, *LNCS*, vol. 9333, pp. 38–54. Springer (2015)
48. Cassar, I., Francalanza, A.: On implementing a monitor-oriented programming framework for actor systems. In: E. Ábrahám, M. Huisman (eds.) Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings, *LNCS*, vol. 9681, pp. 176–192. Springer (2016)
49. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A generic instrumentation tool for Erlang. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA, *Kalpa Publications in Computing*, vol. 3, pp. 48–54. EasyChair (2017)
50. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A suite of monitoring tools for Erlang. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 41–47. EasyChair (2017)
51. Cheikh, A.B., Blein, Y., Chehida, S., Vega, G., Ledru, Y., du Bousquet, L.: An environment for the partrap trace property language (tool demonstration). In: C. Colombo, M. Leucker (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, *LNCS*, vol. 11237, pp. 437–446. Springer (2018)
52. Cohen, S.: JTrek (2000). Developed by Compaq
53. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: C. Dubois, D. Giannakopoulou, D. Méry (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014., *EPTCS*, vol. 149, pp. 79–92 (2014)
54. Colombo, C., Falcone, Y.: First international summer school on runtime verification - as part of the ARVI COST action 1402. In: Y. Falcone, C. Sánchez (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, *LNCS*, vol. 10012, pp. 17–20. Springer (2016)
55. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design **49**(1-2), 109–158 (2016)
56. Colombo, C., Pace, G.J.: Runtime verification using LARVA. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 55–63. EasyChair (2017)
57. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: D.D. Cofer, A. Fantechi (eds.) FMICS 2008, *LNCS*, vol. 5596, pp. 135–149. Springer (2008)
58. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: D.V. Hung, P. Krishnan (eds.) SEFM 2009, pp. 33–37. IEEE Computer Society (2009)
59. DeAntoni, J., Mallet, F.: Timesquare: Treat your models with logical time. In: C.A. Furia, S. Nanz (eds.) TOOLS 2012, *LNCS*, vol. 7304, pp. 34–41. Springer (2012)
60. Decker, N., Harder, J., Scheffel, T., Schmitz, M., Thoma, D.: Runtime monitoring with union-find structures. In: M. Chechik, J. Raskin (eds.) TACAS 2016, *LNCS*, vol. 9636, pp. 868–884. Springer (2016)
61. Decker, N., Leucker, M., Thoma, D.: jUnit^{RV}-adding runtime verification to jUnit. In: G. Brat, N. Rungta, A. Venet (eds.) NFM 2013, *LNCS*, vol. 7871, pp. 459–464. Springer (2013)
62. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. STTT **18**(2), 205–225 (2016)
63. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: S.Y. Shin, J.C. Maldonado (eds.) SAC 2013, pp. 1230–1235. ACM (2013)
64. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Software Eng. **30**(12), 859–872 (2004)
65. Díaz, A., Merino, P., Salmerón, A.: Obtaining models for realistic mobile network simulations using real traces. IEEE Communications Letters **15**(7), 782–784 (2011)
66. Donzé, A.: Breach, A toolbox for verification and parameter synthesis of hybrid systems. In: T. Touili, B. Cook, P.B. Jackson (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, *LNCS*, vol. 6174, pp. 167–170. Springer (2010)
67. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, *LNCS*, vol. 8044, pp. 264–279. Springer (2013)
68. Dou, W., Bianculli, D., Briand, L.: TemPsy-Check: a tool for model-driven trace checking of pattern-based temporal properties. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 64–70. EasyChair (2017)
69. Dou, W., Bianculli, D., Briand, L.C.: A model-driven approach to trace checking of pattern-based temporal properties. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017, pp. 323–333. IEEE Computer Society (2017)
70. Drabek, C., Weiss, G.: DANA – description and analysis of networked applications. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 71–80. EasyChair (2017)
71. Drusinsky, D.: Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. Newnes, Newton, MA, USA (2006)
72. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: T. Bultan, K. Sen (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, pp. 125–135. ACM (2017)
73. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: T. Bultan, K. Sen (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, pp. 372–375. ACM (2017)
74. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. **29**(1), 1:1–1:57 (2020). DOI 10.1145/3355181
75. Ellul, J., Pace, G.J.: Runtime verification of ethereum smart contracts. In: 14th European Dependable Computing Conference, EDCC 2018, Iaşi, Romania, September 10-14, 2018, pp. 158–163. IEEE Computer Society (2018)
76. Falcone, Y.: You should better enforce than verify. In: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, *LNCS*, vol. 6418, pp. 89–105. Springer (2010)

77. Falcone, Y.: Second school on runtime verification, as part of the arvi COST action 1402 - overview and reflections. In: Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, pp. 27–32 (2018). DOI 10.1007/978-3-030-03769-7_3
78. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: M. Broy, D.A. Peled, G. Kalus (eds.) Engineering Dependable Software Systems, *NATO SPS D: Information and Communication Security*, vol. 34, pp. 141–175. IOS Press (2013)
79. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: International Conference on Runtime Verification, pp. 241–262. Springer (2018)
80. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci and Falcone [22], pp. 103–134. DOI 10.1007/978-3-319-75632-5_4
81. Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification CRV 2015. In: E. Bartocci, R. Majumdar (eds.) RV 2015, *LNCS*, vol. 9333, pp. 405–422. Springer (2015)
82. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: Streamlab: Stream-based monitoring of cyber-physical systems. In: I. Dillig, S. Tasiran (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I, *LNCS*, vol. 11561, pp. 421–431. Springer (2019)
83. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper : A runtime verification tool for temporal hyperproperties. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II, pp. 194–200 (2018)
84. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the hennessy-milner logic with recursion. Formal Methods in System Design **51**(1), 87–116 (2017)
85. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: E. Bartocci, Y. Falcone (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, *LNCS*, vol. 10457, pp. 176–210. Springer (2018)
86. Goldberg, A., Havelund, K.: Automated runtime verification with eagle. In: U. Ultes-Nitsche, J.C. Augusto, J. Barjis (eds.) Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2005, In conjunction with ICEIS 2005, Miami, FL, USA, May 2005. INSTICC Press (2005)
87. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: C. Colombo, M. Leucker (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, *LNCS*, vol. 11237, pp. 282–298. Springer (2018)
88. Goubault-Larrecq, J., Lachance, J.: On the complexity of monitoring orchids signatures, and recurrence equations. Formal Methods in System Design **53**(1), 6–32 (2018)
89. Goubault-Larrecq, J., Olivain, J.: A smell of orchids. In: M. Leucker (ed.) Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers, *LNCS*, vol. 5289, pp. 1–20. Springer (2008)
90. Groce, A., Havelund, K., Smith, M.H.: From scripts to specifications: the evolution of a flight software testing effort. In: J. Kramer, J. Bishop, P.T. Devanbu, S. Uchitel (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp. 129–138. ACM (2010)
91. Hallé, S.: When RV meets CEP. In: Y. Falcone, C. Sánchez (eds.) RV 2016, *LNCS*, vol. 10012, pp. 68–91. Springer (2016)
92. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 81–88. EasyChair (2017)
93. Havelund, K.: Runtime verification of C programs. In: K. Suzuki, T. Higashino, A. Ulrich, T. Hasegawa (eds.) Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings, *LNCS*, vol. 5047, pp. 7–22. Springer (2008)
94. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015)
95. Havelund, K., Leucker, M., Reger, G., Stolz, V.: A shared challenge in behavioural specification (Dagstuhl seminar 17462). Dagstuhl Reports **7**(11), 59–85 (2017)
96. Havelund, K., Peled, D.: Bdds on the run. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV, *LNCS*, vol. 11247, pp. 58–69. Springer (2018)
97. Havelund, K., Peled, D.: Efficient runtime verification of first-order temporal properties. In: M. Gallardo, P. Merino (eds.) Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings, *LNCS*, vol. 10869, pp. 26–47. Springer (2018)
98. Havelund, K., Peled, D.: Runtime verification: From propositional to first-order temporal logic. In: C. Colombo, M. Leucker (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, *LNCS*, vol. 11237, pp. 90–112. Springer (2018)
99. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with bdds. In: 2017 Formal Methods in Computer Aided Design (FMCAD), pp. 116–123 (2017)
100. Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic. In: 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, April 10, 2018, pp. 12–13. IEEE (2018)
101. Havelund, K., Reger, G.: Runtime verification logics A language design perspective. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, pp. 310–338 (2017)
102. Havelund, K., Rosu, G.: Monitoring java programs with java pathexplorer. Electr. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001)
103. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: J. Katoen, P. Stevens (eds.) 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, *LNCS*, vol. 2280, pp. 342–356. Springer (2002)
104. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. STTT **6**(2), 158–173 (2004)
105. Havelund, K., Rosu, G.: An overview of the runtime verification tool java pathexplorer. Formal Methods in System Design **24**(2), 189–215 (2004)
106. Jakse, R., Falcone, Y., Méhaut, J., Pouget, K.: Interactive runtime verification - when interactive debugging meets runtime verification. In: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017, pp. 182–193. IEEE Computer Society (2017). DOI 10.1109/ISSRE.2017.19. URL `https://doi.org/10.1109/ISSRE.2017.19`
107. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: M. Glinz, G.C. Murphy, M. Pezzè (eds.) ICSE 2012, pp. 1427–1430. IEEE Computer Society (2012)

108. Kane, A.: Runtime monitoring for safety-critical embedded systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, USA (2015). URL https://dx.doi.org/10.1184/R1/6721376.v1
109. Kane, A., Chowdhury, O., Datta, A., Koopman, P.: A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In: E. Bartocci, R. Majumdar (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, *LNCS*, vol. 9333, pp. 102–117. Springer (2015)
110. Kauffman, S., Fischmeister, S.: Event stream abstraction using nfer: demo abstract. In: X. Liu, P. Tabuada, M. Pajic, L. Bushnell (eds.) Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019, pp. 332–333. ACM (2019)
111. Kauffman, S., Havelund, K., Joshi, R.: nfer - A notation and system for inferring event stream abstractions. In: Y. Falcone, C. Sánchez (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, *LNCS*, vol. 10012, pp. 235–250. Springer (2016)
112. Kauffman, S., Havelund, K., Joshi, R., Fischmeister, S.: Inferring event stream abstractions. Formal Methods in System Design **53**(1), 54–82 (2018)
113. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: J.L. Knudsen (ed.) ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings, *LNCS*, vol. 2072, pp. 327–353. Springer (2001)
114. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: runtime verification of non-synchronized real-time streams. In: H.M. Haddad, R.L. Wainwright, R. Chbeir (eds.) Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, pp. 1925–1933. ACM (2018)
115. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
116. Lourenço, J.M., Fiedor, J., Krena, B., Vojnar, T.: Discovering concurrency errors. In: E. Bartocci, Y. Falcone (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, *LNCS*, vol. 10457, pp. 34–60. Springer (2018)
117. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Serbanuta, T., Rosu, G.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: B. Bonakdarpour, S.A. Smolka (eds.) RV 2014, *LNCS*, vol. 8734, pp. 285–300. Springer (2014)
118. Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. IEEE Trans. Software Eng. **37**(4), 486–508 (2011)
119. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012)
120. Milewicz, R., Vanka, R., Tuck, J., Quinlan, D., Pirkelbauer, P.: Lightweight runtime checking of C programs with RTC. Comput. Lang. Syst. Str. **45**, 191–203 (2016)
121. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Form. Methods Sys. Des. **51**(1), 31–61 (2017)
122. Navabpour, S., Joshi, Y., Wu, C.W.W., Berkovich, S., Medhat, R., Bonakdarpour, B., Fischmeister, S.: RiTHM: a tool for enabling time-triggered runtime verification for C programs. In: B. Meyer, L. Baresi, M. Mezini (eds.) ESEC/FSE 2013, pp. 603–606. ACM (2013)
123. Olivain, J., Goubault-Larrecq, J.: The orchids intrusion detection tool. In: K. Etessami, S.K. Rajamani (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, *LNCS*, vol. 3576, pp. 286–290. Springer (2005)
124. Pastore, F., Mariani, L.: AVA: supporting debugging with failure interpretations. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, pp. 416–421. IEEE Computer Society (2013)
125. Pastore, F., Mariani, L., Goffi, A., Oriol, M., Wahler, M.: Dynamic analysis of upgrades in C/C++ software. In: 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012, pp. 91–100. IEEE Computer Society (2012)
126. Pastore, F., Mariani, L., Goffi, A., Oriol, M., Wahler, M.: RADAR: dynamic analysis of upgrades in C/C++ software. In: H. Chockler, D. Kroening, L. Mariani, N. Sharygina (eds.) Validation of Evolving Software, pp. 85–105. Springer (2015)
127. Piel, A.: Reconnaissance de comportements complexes par traitement en ligne de flux d'événements. (Online event flow processing for complex behaviour recognition). Ph.D. thesis, Paris 13 University, Villetaneuse, Saint-Denis, Bobigny, France (2014). URL https://tel.archives-ouvertes.fr/tel-01262245
128. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Tipex: A tool chain for timed property enforcement during execution. In: E. Bartocci, R. Majumdar (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, *LNCS*, vol. 9333, pp. 306–320. Springer (2015)
129. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods in System Design **45**(3), 381–422 (2014)
130. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings, *LNCS*, vol. 4085, pp. 573–586. Springer (2006). DOI 10.1007/11813040_38
131. Pnueli, A., Zaks, A.: On the merits of temporal testers. In: O. Grumberg, H. Veith (eds.) 25 Years of Model Checking - History, Achievements, Perspectives, *LNCS*, vol. 5000, pp. 172–195. Springer (2008)
132. Rapin, N.: Reactive property monitoring of hybrid systems with aggregation. In: Y. Falcone, C. Sánchez (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, *LNCS*, vol. 10012, pp. 447–453. Springer (2016)
133. Rapin, N.: ARTiMon monitoring tool, the time domains. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 106–122. EasyChair (2017)
134. Raszyk, M., Basin, D., Traytel, D.: Multi-head monitoring of metric dynamic logic. In: D.V. Hung, O. Sokolsky (eds.) ATVA 2020, *LNCS*, vol. 12302. Springer (2020). To appear.
135. Raszyk, M., Basin, D.A., Krstic, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Y. Chen, C. Cheng, J. Esparza (eds.) ATVA 2019, *LNCS*, vol. 11781, pp. 151–170. Springer (2019)
136. Reger, G.: An overview of MarQ. In: Y. Falcone, C. Sánchez (eds.) RV 2016, *LNCS*, vol. 10012, pp. 498–503. Springer (2016)
137. Reger, G.: A report of RV-CuBES 2017. In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 1–9. EasyChair (2017)
138. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: monitoring at runtime with QEA. In: C. Baier, C. Tinelli (eds.) TACAS 2015, *LNCS*, vol. 9035, pp. 596–610. Springer (2015)
139. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification - CRV 2016. In: Y. Falcone, C. Sánchez (eds.) RV 2016, *LNCS*, vol. 10012, pp. 21–37. Springer (2016)
140. Reger, G., Havelund, K. (eds.): RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, *Kalpa Publications in Computing*, vol. 3. EasyChair (2017)

141. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: E. Ábrahám, K. Havelund (eds.) TACAS 2014, *LNCS*, vol. 8413, pp. 357–372. Springer (2014)
142. Renard, M., Rollet, A., Falcone, Y.: GREP: games for the runtime enforcement of properties. In: N. Yevtushenko, A.R. Cavalli, H. Yenigün (eds.) Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings, *LNCS*, vol. 10533, pp. 259–275. Springer (2017). DOI 10.1007/978-3-319-67549-7_16
143. Riganelli, O., Micucci, D., Mariani, L.: Policy enforcement with proactive libraries. In: 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017, pp. 182–192. IEEE Computer Society (2017)
144. Riganelli, O., Micucci, D., Mariani, L.: Increasing the reusability of enforcers with lifecycle events. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV, *LNCS*, vol. 11247, pp. 51–57. Springer (2018)
145. Salmerón, A.: Model checking techniques for runtime testing and qos analysis. Ph.D. thesis
146. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: C. Colombo, M. Leucker (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, *LNCS*, vol. 11237, pp. 138–163. Springer (2018)
147. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). DOI 10.1007/s10703-019-00337-w
148. Santos, J.F., Jensen, T., Rezk, T., Schmitt, A.: Hybrid typing of secure information flow in a javascript-like language. In: P. Ganty, M. Loreti (eds.) Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers, *LNCS*, vol. 9533, pp. 63–78. Springer (2015)
149. Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime analysis with R2U2: A tool exhibition report. In: Y. Falcone, C. Sánchez (eds.) RV 2016, *LNCS*, vol. 10012, pp. 504–509. Springer (2016)
150. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: G. Heiser, W.C. Hsieh (eds.) 2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012, pp. 309–318. USENIX Association (2012)
151. Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., Zadok, E.: Interaspect: aspect-oriented instrumentation with GCC. Formal Methods in System Design **41**(3), 295–320 (2012)
152. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: G. Reger, K. Havelund (eds.) RV-CuBES 2017, *Kalpa Publications in Computing*, vol. 3, pp. 164–173. EasyChair (2017)
153. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: Sanitizing for security. CoRR **abs/1806.04355** (2018)
154. Soueidi, C., Kassem, A., Falcone, Y.: BISM: bytecode-level instrumentation for software monitoring. In: Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings, pp. 323–335 (2020). DOI 10.1007/978-3-030-60508-7_18
155. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: S. Khurshid, K. Sen (eds.) Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers, *LNCS*, vol. 7186, pp. 193–207. Springer (2011). DOI 10.1007/978-3-642-29860-8_15. URL `https://doi.org/10.1007/978-3-642-29860-8_15`
156. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013, pp. 48–62. IEEE Computer Society (2013)
157. Ulus, D.: Montre: A tool for monitoring timed regular expressions. In: R. Majumdar, V. Kuncak (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, *LNCS*, vol. 10426, pp. 329–335. Springer (2017)
158. Ulus, D.: Sequential circuits from regular expressions revisited. CoRR **abs/1801.08979** (2018)
159. Ulus, D.: Online monitoring of metric temporal logic using sequential networks. CoRR **abs/1901.00175** (2019)
160. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: A. Legay, M. Bozga (eds.) Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, Florence, Italy, September 8-10, 2014. Proceedings, *LNCS*, vol. 8711, pp. 222–236. Springer (2014)
161. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Online timed pattern matching using derivatives. In: M. Chechik, J. Raskin (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, *LNCS*, vol. 9636, pp. 736–751. Springer (2016)
162. van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory errors: The past, the present, and the future. In: D. Balzarotti, S.J. Stolfo, M. Cova (eds.) Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings, *LNCS*, vol. 7462, pp. 86–106. Springer (2012)
163. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empirical Software Engineering **21**(3), 811–853 (2016)
164. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: J. Torrellas, S. Chatterjee (eds.) Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006, pp. 137–146. ACM (2006)
165. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. **32**(2), 93–110 (2006)