Computer Science Review 50 (2023) 100594

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Review article

Uncertainty in runtime verification: A survey

Rania Taleb [a], Sylvain Hallé [a,*], Raphaël Khoury [b]

[a] Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Canada
[b] Université du Québec en Outaouais, Canada

ARTICLE INFO

Article history:
Received 25 April 2023
Received in revised form 21 August 2023
Accepted 26 August 2023
Available online xxxx

ABSTRACT

Runtime Verification can be defined as a collection of formal methods for studying the dynamic evaluation of execution traces against formal specifications. Aside from creating a monitor from specifications and building algorithms for the evaluation of the trace, the process of gathering events and making them available for the monitor and the communication between the system under analysis and the monitor are critical and important steps in the runtime verification process. In many situations and for a variety of reasons, the event trace could be incomplete or could contain imprecise events. When a missing or ambiguous event is detected, the monitor may be unable to deliver a sound verdict. In this survey, we review the literature dealing with the problem of monitoring with incomplete traces. We list the different causes of uncertainty that have been identified, and analyze their effect on the monitoring process. We identify and compare the different methods that have been proposed to perform monitoring on such traces, highlighting the advantages and drawbacks of each method.

Contents

* Corresponding author.
E-mail address: shalle@acm.org (S. Hallé).

https://doi.org/10.1016/j.cosrev.2023.100594

1. Introduction

Runtime Verification (RV) or Runtime Monitoring has gained an increasing interest in the recent years [13]. It can be defined as the process of observing the behavior of a running system and determining whether the execution under study is compliant with the expected behavior of the system, and detect any violations [3]. The running behavior is represented by the execution trace (the sequence of events produced by the system). The expected behavior is usually specified as a set of rules or formal properties that must be obeyed. A property generally involves conditions on the sequence of events, as well as the data inside these events.

Unlike other verification techniques, such as testing [14], and other formal verification methods such as model checking [15] and theorem proving [16], which are typically performed offline, RV can be performed online while the system is executing. Rather than relying on a model of the system and its environment, which can be extremely complicated and potentially result in a state explosion problem, RV works directly with the actual system; in counterpart, it typically analyzes a single execution trace at a time.

The process of collecting the trace of events and presenting it to the monitor is critical. Events can be collected from various sources such as the events gathered from system instrumentation [17–22] or external values measured and recorded by sensor devices [23]. Moreover, there is no general convention on what format the events should take in the trace. Many notations and formats can be used to represent events depending on the monitoring framework employed [24].

Regardless of the variety of event sources, most of the RV approaches assume that the monitor has complete and error-free access to the trace of events against which to evaluate a given property [25–27]. However, there are multiple situations where this assumption does not hold, such as in the case of incorrect system instrumentation, imprecise measurements, sampling techniques applied in RV to control overhead, and misconfiguration of data access control policies, among others. In this respect, a recent Dagstuhl seminar report has emphasized the importance of dealing with incomplete, imprecise, and faulty sources of events [28], as did a recent survey of challenges related to RV [29]. Ignoring the fact that incomplete and imprecise events might have occurred gives poor monitoring results. A sound and complete monitor should have a reasonable level of certainty about the content of the underlying trace that allows it to produce a conclusive verdict.

A variety of works have tackled the problem of RV with incomplete, uncertain or missing information in the past decade. However, these approaches vary greatly in several dimensions of the problem, which makes them difficult to compare. The majority of these techniques rely on recovering lost events to reach a sound and meaningful verdict. This recovery is accomplished through various means: constructing a probabilistic model (Sections 4.3.1, 4.3.2, 4.3.3), filling gaps using all possible replacements (Sections 4.1.1, 4.1.3), representing missing events with symbols (Sections 4.1.2, 4.2.1, 4.2.2, 4.2.5, 4.3.1, 4.3.2, 4.3.3), or denoting a sequence of events as an interval (Sections 4.2.3, 4.2.4). Certain techniques aim to come up with a conservative approximation of all the possible verdicts (Sections 4.1.1, 4.1.2). Others provide a probability of satisfaction (Sections 4.3.1, 4.3.2, 4.3.3), and some methods yield one single verdict (Sections 4.1.3, 4.2.1, 4.2.2, 4.2.3, 4.2.4, 4.2.5, 4.2.6). Additionally, several propose new specification languages (Sections 4.1.2, 4.2.2, 4.2.4, 4.2.6) or extend existing ones with operators that allows the monitoring of some properties in the presence of incomplete events and the production of sound verdicts in some situations (Sections 4.2.1, 4.2.5). Regardless the reasons why incomplete or uncertain data may occur, the way in which a "perfect" trace into an incomplete one varies among different works. Some focus solely on accounting for missing events (Sections 4.2.1, 4.2.6, 4.3.1, 4.3.2, 4.3.3), or imprecise events exclusively (Section 4.2.5), or both missing and imprecise events (Sections 4.1.1, 4.2.2). Others address unordered events alone (Section 4.2.4), or both missing and unordered events (Section 4.1.3). Specific methods also handle imprecise timestamps (Section 4.2.3), while some broaden their scope to accommodate missing events, imprecise events, and imprecise timestamps (Section 4.1.2).

In fact, the problem of RV under uncertainty being relatively recent, each of the contributions presents its approach in isolation, without really discussing its relation with other similar

works. We are therefore confronted with an extremely fragmented vision of the state of the art on the question, which has the effect of making it difficult to identify the avenues of research on which work remains to be done. This is the goal of this survey, where we describe and synthesize different approaches from the literature that seek to employ formal, statistical, and other techniques to handle RV for systems with incomplete traces.

The remainder of the paper is structured as follows. In Section 2, we review the stages of an RV process, the different types of events and their representation, and the main specification languages used for RV. In Section 3, we describe and classify different situations that alter a source of events and cause data loss. Then, we discuss two important points: first, in Section 4 we describe and classify different approaches from the literature that account for the problem of RV with incomplete data; second, in Section 5, we discuss the features and limitations of each approach with respect to the other approaches. Finally, Section 6 draws conclusions and identifies directions for future work on the subject.

2. Overview of runtime verification

Runtime Verification [13] serves as a useful complement to offline verification techniques such as model checking and theorem proving, as well as partial solutions like testing and debugging. While model checking explores all possible system states using a formal model and theorem proving establishes correctness through mathematical proofs, RV operates differently. It lacks a predefined model and does not statically analyze the system. Instead, it draws conclusions solely from observed executions. Consequently, RV's conclusions are confined to what it has witnessed at runtime, in contrast to model checking's exhaustive exploration of potential states. Hence, unlike formal techniques that can prove correctness, RV can only be used to detect problems within the system and emit a conclusive verdict.

By combining the exhaustive nature of offline verification methods with the application to actual program traces as seen in testing and debugging, RV provides the best of both worlds. However, RV's results possess an intriguing yet challenging characteristic: an RV monitor detecting a violation is undoubtedly informative. However, the absence of a violation observed by a monitor does not allow us to conclude that the entire system is correct. This aspect underscores the nuanced nature of RV's role in verification. In contrast, model checking can unequivocally confirm correctness for all states explored, while theorem proving establishes correctness based on rigorous mathematical reasoning. In essence, RV's utility lies primarily in its ability to uncover issues and anomalies at runtime, offering valuable insights into system behavior as it executes. To optimally position RV within the realm of formal verification techniques, it is imperative to recognize its unique strengths and limitations in comparison to model checking and theorem proving, appreciating its emphasis on runtime behavior analysis rather than exhaustive state exploration or mathematical proofs of correctness.

One challenge of integrating RV into a system is managing the resulting runtime overhead which can arise from a range of factors, such as the monitor invocation, the computation and evaluation of property predicates based on the program's state, potential performance slowdown due to program instrumentation and trace extraction, and potential interference between the program and monitor as the monitor may share resources with the program.

Another challenge in RV is the source and type of events available to the monitor. An event is not necessarily an observation detected during system execution. It can refer to a wide variety of phenomena outside the system, such as an event recorded by the environmental sensors that capture data from the system's surrounding environment (temperature, humidity, pressure, or light), and can be of numerical type (e.g. integer, decimal, etc.). External devices such as cameras and microphones can also capture events of type image and audio clip. Messages transferred through a network, such as HTTP requests, can also be considered as events of type text (strings).

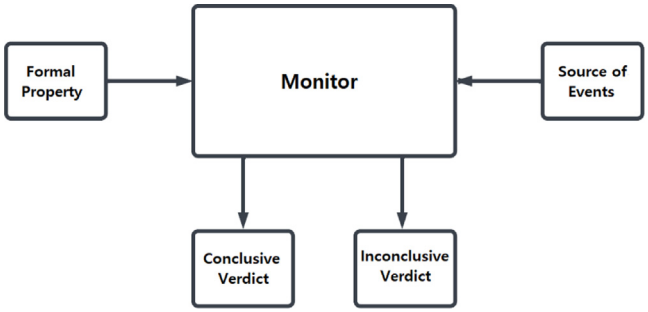

Fig. 1. RV Setup.

A specific RV problem is defined by the format used to represent events in the traces produced by a system, as well as the specification language that represents conditions (also called "properties") over these events. To some extent, these different combinations have been shown to be translatable into each other [30], although with some possible loss when the expressiveness of the formats differ. In this section, we aim to describe the stages of RV and define the notions of events, traces, and properties.

2.1. Stages of the RV

A typical RV setup, such as illustrated in Fig. 1, consists of creating a monitor from a specification, extracting a trace from the execution of the target system, and evaluating this trace against the specified property. A specification property is a formal statement that defines the desired behavior of the system. It specifies what the system is expected to do or not do in response to different inputs and under varying circumstances. The property typically consists of a set of rules or constraints that must hold true for the system to meet its intended purpose, and can be expressed in various formal languages, such as temporal logic or automata.

2.1.1. Synthesizing the RV monitor from a property

Depending on the specification language used to express it, a property may not directly provide an algorithm to evaluate it on a trace of events. This is the case, for example, of Linear Temporal Logic (LTL) [1], an extension of propositional logic that allows assertions on the ordering of events in a sequence. Therefore, to apply run-time verification to a property written in a formal notation, it is necessary in many cases to first create a *monitor* that can concretely evaluate the property.

For the case of LTL, Bauer et al. propose a step-by-step method that takes an LTL property φ as an input and produces a deterministic finite state machine (FSM) as output [25]. Fig. 2 illustrates the steps. The first step is to convert the LTL formula into a Non-deterministic Büchi Automaton (NBA) using one of several possible algorithms [31–38]. An NBA is a type of automaton that accepts infinite sequences of states, and it can be used to represent all the possible executions that satisfy some LTL property. The next step is to simplify the NBA by removing any redundant states and transitions. This is done using algorithms such as the subset construction or the power set construction. The third step

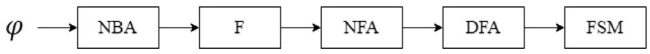

Fig. 2. Steps required to generate an FSM from an LTL formula φ.

is to convert the reduced NBA into a Non-deterministic Finite Automaton (NFA), which is a type of automaton that accepts a finite sequence of states. This is done by removing the acceptance condition from the NBA and converting it into a transition system. The fourth step is to convert the NFA into a Deterministic Finite Automaton (DFA), which is a type of automaton that has a unique transition for each input symbol and state. The final step is to map the DFA to an FSM by assigning state variables to each state and defining the transition function that determines the next state based on the current state and input variables.

2.1.2. System instrumentation

The system instrumentation step is a crucial stage in RV during which the monitor is able to connect with the system that produces the events that need to be observed and processed [39]. In the case of a software system, instrumentation can be done at the source code level [21], by adding extra code instructions to the source files before compilation to track the execution of particular software components and to output an execution trace that can be fed to the monitor. A similar operation can also be done on compiled code, at the binary level [40].

However, although early RV works have focused on instrumented software systems, over the years the scope of what constitutes a possible source of events has been expanded. For example, *system logs* can provide valuable insights into the behavior of the system, including errors, warnings, and other events that occur during its execution; thus, one can collect data on various system parameters, such as CPU usage, memory usage, and disk I/O, which can be analyzed to identify potential issues or areas for improvement.

In an even broader way, one can consider systems whose sources of events come from even more diverse sources. As a matter of fact, any event or data point that is relevant to the behavior of the system can be instrumented and monitored for analysis, depending on the specific requirements and goals of the RV framework. For example, in software systems that interact with users, monitoring and logging user interactions can provide valuable insights into how users interact with the system, and can help identify potential issues or areas for improvement. In distributed systems or networked applications, monitoring network traffic can provide insights into the behavior of the system, and can help identify issues related to performance, security, or communication between system components. In systems that interact with physical sensors, monitoring and analyzing sensor data can provide insights into the behavior of the system, and can help identify issues related to sensor accuracy, calibration, or data processing.

2.1.3. Analyzing system execution

Following instrumentation, the retrieved events are transmitted to the monitor for analysis. This process is frequently known as execution analysis. The monitor examines the trace one event at a time. The monitoring can happen either *offline*, where the execution trace is previously kept in a log and supplied to the monitor, or *online*, where the event analysis is performed during the execution in a lock-step manner [41].

The monitor interacts with the system by emitting a *verdict* for each event consumed, which indicates the status of the property at that point of the execution (i.e. considering the last event as well as all preceding ones). In the simplest case, the verdict domain could be $\mathbb{B}_2 = \{\bot, \top\}$ where $\top$ represents the true verdict indicating that the property is satisfied and $\bot$ represents the negative verdict indicating that the property is violated. However, most RV systems aim to provide a more fine-grained result and use verdict domains containing three or more values. A common domain is $\mathbb{B}_3 = \{\bot, ?, \top\}$ where "?" means that there is not enough information to conclude either satisfaction or violation and the monitor is not able to produce a conclusive verdict in the current state of the system. Finer-grained verdicts with 4 and even 5 truth values have also been considered [42].

The monitor can also communicate with the system by sending *feedback* so that suitable corrective actions can be taken if the property is violated. This is a field of study in its own, known as *Runtime Enforcement* [43–45] which extends the field of RV in that it aims to modify the trace by deleting events, inserting or modifying the events to *correct* any illicit behavior present in the trace, rather than simply detect it. The monitor thus acts as a transducer, replacing the original, possibly invalid execution with an alternative, newer execution that provably respects the desired property.

The above stages of RV have been applied in numerous and various situations: monitoring programs to check if its execution satisfies a property [46,47]; monitoring and recovery of web service applications [48–50] where the source of events to monitor are web services or other forms of web based implementations; monitoring of driving emissions from a vehicle [51]; detection of bugs in video games [52]; verification of the behavior of aerial drones [53], among others.

2.2. Events and event types

Any kind of observation about a system is called an *event*. Values read and recorded by sensor devices, regardless of whether they are strings or numbers or any other type, are events. Inner actions performed in a software system such as returning the results of a web search, adding a user to a database, reading or writing to a file, snapshots of the system's status taken at regular intervals, etc., can also be considered as events. What is called a *trace* is the (linear) succession of events measured or produced by the execution of the system. There is a variety of formats and notations that can be used to represent the events [24]. In this section, we enumerate the most common types of events and show how each event type could be represented.

2.2.1. Atomic symbols

In the simplest case, an event is a name for something that can happen such as *openFile* or *closeFile*, or carrying a value such as a string or number or a Boolean from the domain $\mathbb{B}_2 = \{\top, \bot\}$. Although this is the simplest and least structured form of event, this simple notation has been used by several works in the literature [2,5,11,54].

2.2.2. CSV events

While atomic events are appropriate in some situations, in many cases, events require to be represented in a more structured form. A first possibility is to represent an event as a *tuple* composed of attributes and values such as data in a CSV (Comma-Separated Values) file. Indeed, CSV events can be likened to a form of tuple: each line of the file is taken as an event, and each element of the line corresponds to the value of an attribute of this event. An example of such CSV trace is shown in Fig. 3. In this example, the first "line" provides the name of four attributes; the remaining lines represent one event each, containing the value corresponding to each attribute. Note that this notation allows events to have empty values for some attributes.

Tuple-based events are commonly used; for example, Havelund et al. presented a benchmark for evaluating RV tools

```
event, map, collection, iterator
updateMAp, 6750210, ,
createColl, 6750210, 2081191879,
createIter, , 2081191879, 910091170
useIter, , , 910091170
updateMap, 1183888521, ,
```

Fig. 3. An example of a trace of CSV events.

in which traces of events are represented in CSV formats [55]. Similarly, during the latest RV competition, CSV files were used to keep track of Java operations on maps. Earlier works have even suggested performing the task of RV on tuple events by translating it into the evaluation of an equivalent database query [56]. The CSV or tuple format is also used to represent events in several other RV approaches such that the approach of Ayesha et al. [57], Jonas et al. [58], and Vikas et al. [59].

2.2.3. XML and JSON events

XML, which stands for eXtensible Markup Language [60], is typically associated to web services [61]. Data is expressed in XML as a tree structure. One common way to structure events in XML is to define a root element that contains one or more child elements, each representing a specific event. Each event element can contain attributes that describe the event, such as a timestamp, an event type, or any other relevant metadata. The content of each event element can include any additional data associated with the event, such as event parameters or payload data. For example, consider a simple XML representation of a sensor reading event:

```
<sensor-data>
  <reading timestamp=‘‘2023-02-27T10:30:00’’ type=‘‘temperature’’>
    <value unit=‘‘Celsius’’>25</value>
  </reading>
</sensor-data>
```

In this example, the sensor-data element is the root element that contains a single reading element representing a sensor reading event. The reading element contains two attributes (timestamp and type) that provide metadata about the event. The value element contains the actual sensor reading value (25) and an attribute (unit) specifying the unit of measurement. The “tags” are the syntactical feature used to represent elements in a text file.

One of the key benefits of using XML (Extensible Markup Language) is that it is a widely supported and standardized format that can be parsed by many existing libraries in various programming languages. The XES format is an IEEE effort to standardize the representation of event data in XML [62]. Many RV frameworks used XML-based format to represent events such as the LogFire framework [63], JRec runtime monitoring framework for web services [64], AXML runtime monitoring framework of XML documents [65], and XMonitor runtime monitoring framework [66].

JavaScript Object Notation (JSON) is also used to represent structured data. Rather using “tags”, JSON uses a simpler syntax consisting of key–value pairs, arrays, and nested objects to represent an event. The above example can be represented in JSON as shown in Fig. 4. By convention, the “@” symbol is used to represent attributes, and “#” symbol is used to represent the text content.

Some RV frameworks use JSON to represent events, such as the FLINT [67], Umbral [68], Varan [69], Panda [70], and Medusa [71].

```
{
  ‘‘sensor-data’’: {
    ‘‘reading’’: {
      ‘‘@timestamp’’: ‘‘2023-02-27T10:30:00’’,
      ‘‘@type’’: ‘‘temperature’’,
      ‘‘value’’: {
        ‘‘@unit’’: ‘‘Celsius’’,
        ‘‘#text’’: 25
      }
    }
  }
}
```

Fig. 4. A sensor event represented in JSON.

	0	1	2	3
x	2	3	2	1
y^1	-	3	2	-
y^2	-	2	1	3
y^3	-	4	3	-
y^4	4	-	2	4

Fig. 5. An event of type snapshot.

2.2.4. Events as predicates

An even more flexible way of representing events consists of modeling them as a set of *predicates* [6]. Formally, given a set of objects S, a predicate can be defined as a function $p : S^n \to \mathbb{B}_2$, where the value n is called the *arity* of the predicate. Given a fixed set of predicates $p_1, \ldots, p_m$ (each with a possibly different arity), an event can be then be represented as a function that defines the value of each predicate for each possible argument.

As an example, consider the simple situation where the set of objects is made of two light bulbs $S = \{a, b\}$, and the predicate $on : S \to \mathbb{B}_2$ which represents the fact that a light bulb is on. A possible event in this context could be $\{on(a) = \top, on(b) = \bot\}$, which indicates the situation where light bulb a is on and b is off. A trace is just a succession of such events, where the definition of each predicate may obviously change from one event to the next, thus representing their varying data content.

This basic model can be extended to allow predicates with more than one argument, and also predicates where each argument may be taken from a different set. One can see that this representation subsumes (i.e. is more general than) the previous formats, as it is relatively straightforward to represent tuples or nested structures using a set of appropriately defined predicates.

2.2.5. Snapshots

So far, all event types considered consist of individual data units that represent a single “state” or “action”. However, events may conflate multiple such states or actions into single data structure, possibly loosing information about their actual content and ordering in the process. We then have snapshots of these events [4]. Fig. 5 represents a snapshot of two data variables recorded by *life data recorder* (LDR), a device that records updates to a set of variables generated by a medical device.

The snapshot in the figure is composed of four “frames” recording the variations in the values of two variable x (that occurs one time per frame) and y (that occurs at most four times per frame, so a dash entry means no value recorded for y). These values are represented as a snapshot, since the knowledge about the exact moment where x changed its value with respect to the multiple changes of y is lost. Consequently, each frame recorded

is an abstract representation of several traces of events, where each event is a tuple (x, y). Formally, one possible trace of the variable updates that happen between frame 0 and frame 1 of Fig. 5 can be represented as

$$(2, 4) \xrightarrow{x} (3, 4) \xrightarrow{y} (3, 3) \xrightarrow{y} (3, 2) \xrightarrow{y} (3, 4)$$

if the value of x changes before any change of y; another trace can be

$$(2, 4) \xrightarrow{y} (2, 3) \xrightarrow{x} (3, 3) \xrightarrow{y} (3, 2) \xrightarrow{y} (3, 4)$$

if the value of x changes between the first and second change of y.

2.3. Specification languages

The desired or correct behavior of the system can be specified as a set of specification properties. Each specification property is an expression represented using one of several specification languages [39]. In the following, we describe some of the classical ways of representing a property presented in the literature.

2.3.1. Regular expressions

Since the correct execution of a system is often related to the possible ordering in which events are allowed to be observed, a first natural way of expressing properties is to consider them as patterns that must be matched against a sequence of symbols. To this end, regular expressions are a popular declarative language for describing sets of strings [39].

A regex comprises a sequence of characters describing a search pattern in a text; a typical regex mixes raw symbols (which must be matched as is) with special characters that can be used to represent multiple alternatives or a form of repetition. For instance, a period (“.”) matches any character, while a range (“[ ]”) matches any of the characters contained within the brackets. In addition, quantifier characters can be affixed to a symbol to indicate that the match may occur a variable number of times. Thus, “x?” indicates that x can be observed zero or one time, while “x+” indicates that x may be present at least once. Additionally, grouping characters such as “( )” create a sequence or sub-expression. Finally, alternation characters such as “|” represent the logical OR operator, so that “x | y” indicates that either x or y must be observed.

Regexes can be used to describe a regular language pattern and express a property. As an example, consider the policy stating that a red light should be immediately followed by a green light. The language of this pattern is a collection of strings over the alphabet $\Sigma = \{green, yellow, red\}$. Using regular expression operators, this can be expressed as follows:

$$(\text{green} \mid \text{yellow})^* \text{ red green}^+(\text{green} \mid \text{yellow})^*$$

Examples of monitors accepting regular expressions as their specifications include JavaMOP [72] and SEQ.OPEN [73].

2.3.2. Finite-state automata

A finite-state automaton [74] is a computational model used to describe the behavior of a system that can be in one of a finite number of states, and can transition between those states in response to some input. Formally, it can be defined as a quadruplet $M = \langle \Sigma, S, s_0, \delta, S_F \rangle$ where Σ is the set of input characters, S is a set of states, $s_0 \in S$ is the initial state, $\delta : S \times \Sigma \to S$ is the transition function, and $S_F \subseteq S$ is the set of final or accepting states. For every input, the automaton moves from the current state to the next state using the transition function and it ends in one of the final states. Since the transition function admits at most a single next state given any state and input symbol, the automaton is called a Deterministic Finite Automata (DFA) [75]. The automaton is called non-deterministic if the transition function is replaced by a transition relation.

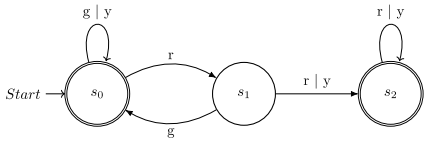

Fig. 6. A finite-state automaton representing the traffic lights property.

Fig. 6 represents the traffic light property using a finite state automaton. Here, s_0 is the initial state and both s_1 and s_2 are final states. The automaton transitions from s_0 to s_1 when encountering a red light event (r) where it should check whether the next input event is g or not. If g appears, the automaton returns to s_0, else if a non-green event appears, then the automaton moves to the final state s_2 and is stuck there producing the same output until the end of the input trace.

An expansion of NFA is the Probabilistic Automaton (PA) [76] that incorporates the likelihood of a particular transition into the transition function, resulting in a transition matrix. The class of languages recognized by probabilistic automata is referred to as stochastic languages, which encompasses regular languages as a subset. The number of stochastic languages is incalculable. In contrast to DFA and NFA, PA employs a weighted set or vector of next states. These weights must total 1, representing probabilities, which renders it a stochastic vector.

2.3.3. LTL: Linear temporal logic [1]

An alternate way of specifying conditions on sequences of events is to turn to logic-based notations. Linear Temporal Logic [1] is one such notation; it is built up from a finite set of propositional variables AP, over which expressions can be constructed using the logical operators ($\neg$, $\vee$ and $\wedge$), and the temporal modal operators (**G**, **F**, **X** and **U**). These operators are called “future time”, as they express conditions that hold from some starting point in a sequence and for subsequent events. If φ represents a condition, the expression $\mathbf{G}\,\varphi$ for example, stands for *globally* and means that a formula φ must hold *globally*, i.e. for every suffix of the current trace. On the other hand, $\mathbf{F}\,\varphi$ stands for *eventually* and stipulates that φ should hold at some point in the future. The expression $\mathbf{X}\,\varphi$ stands for *next*, meaning that φ should hold in the suffix of the trace starting from the *next* event. Finally, the binary operator **U** stands for *until*; the expression $\varphi_1\,\mathbf{U}\,\varphi_2$ means that φ_1 has to hold at least until φ_2 becomes true, and that φ_2 must hold at some point in the future. Several operators can be combined to represent complex conditions on the accepted ordering of events in a trace, such as $\mathbf{G}\,\neg a \wedge \mathbf{F}\,b$, stating that a should never hold and b must finally hold. The traffic light property *“a red light should be immediately followed by a green light”* can be expressed in LTL as follows: $\mathbf{G}(red \rightarrow (\mathbf{X}\,green))$.

Several efforts have been made to augment LTL with quantitative operators that can represent quantitative (metric) real-time properties that are beyond the scope of classical LTL. Given the plethora of these logics, we will only emphasize the significant ones. For a more comprehensive explanation, readers can refer to the cited sources. Metric Temporal Logic (MTL) [77] is the most extensively scrutinized and investigated real-time extension of LTL. As MTL holds significant prominence among other extensions, it will be explained in greater detail in the next section.

Past-LTL [1] extends LTL with temporal operators that refer to past events, allowing expressing properties such as “a has always

been true in the past". Interval LTL [78] also extends LTL with operators that allow specifying properties over intervals of time, such as "a holds for at least k time units within every interval of length n". Probabilistic LTL [79] where probabilistic operators are introduced into LTL, allowing expressing properties with a degree of uncertainty, such as "with probability p, eventually a happens". Quantified LTL [80] where LTL is extended with quantifiers, allowing expressing properties over subsets of the state space, such as "for all states satisfying condition C, A holds eventually". LTL-FO$^+$ extends LTL with quantifiers on data values inside events [50]. Finally, TK-LTL [81] which extends the semantics of LTL with several syntactic structures aimed providing a quantitative evaluation of different aspect of the trace.

2.3.4. MTL: Metric temporal logic

MTL [77,82] is a propositional bounded-operator logic, which is an extension of LTL with timing constraints. Temporal operators (such as 'until', 'next', and'since') are augmented with time references. The **U** operator of LTL is replaced with $\mathbf{U}_I$, where I is an interval of reals with endpoints in $\mathbb{N} \cup \{\infty\}$. MTL can express deadline properties, meaning that the system is required to react within a specified time-frame after a particular action takes place. For example, consider the property that *"every alarm is followed by a shutdown event in 10 s unless all clear is sounded first"*. It is expressed in MTL as: $\square(alarm \rightarrow (\Diamond_{(0,10)} allClear \vee \Diamond_{\{10\}} shutdown))$, where $\square$means *always*, $\Diamond$ means *eventually*, $(0, 10)$ means *'within 10 seconds'* and $\{10\}$ means *'in exactly 10 seconds'*.

MTL can be applied to linearly ordered time domains, which may be represented as discrete, dense, or continuous. The interpretation of MTL varies depending on the selected time flow, and its semantics may change accordingly. For example, suppose that $f : \mathbb{R}^+ \rightarrow 2^{\Sigma}$ is a mapping from a real-time point $t \in \mathbb{R}^+$ to the set of propositions holding at time t. Semantically, in a dense time, we have that $f \models \varphi_1 \cup_I \varphi_2$ if $\exists t \in I$ such that $f^t \models \varphi_2$ and $\forall t' \in (0, t) : f^{t'} \models \varphi_1$, where $f^t(s) = f(t + s)$. MTL can also represent the trace as a sequence of timed words (a time word σ is a finite or infinite word $(t_0, a_0)(t_1, a_1) \ldots \in (\mathbb{R}^+ \times \Sigma)$, where the sequence of t_i is strictly monotonic and non-zero). The semantics in this case can be as follows: $\sigma[i] \models \varphi_1 \cup_I \varphi_2$ if and only if $\exists j \geq i$ such that $\sigma[j] \models \varphi_2, (t_j - t_i) \in I$ and $(\forall i \leq k < j)\sigma[k] \models \varphi_1$.

2.3.5. LOLA: Logic of linear arithmetic

LOLA [83] is a temporal logic-based language that allows users to specify temporal properties over streams using various logical operators, such as conjunction, disjunction, implication, and negation. A stream of events is the same as the trace of events used in other languages, however a stream can be thought of as an infinite sequence of real data values continuously generated and consumed. LOLA is a computation language that accepts a specification in the form of a set of stream equations using typed stream variables. The output streams are computed from a given set of input streams. It has been shown that the expressiveness of LOLA exceeds that of FSM, LTL and MTL (described in Sections 2.3.2 and 2.3.3) because it can handle quantitative constraints over real-valued variables. A stream can be computed using values from other streams by using arithmetic operators, logical operators (such as $\wedge$, $\vee$...), temporal operators (such as *Until*...) and other operators to combine streams. It could also allow a stream to be defined by referring to the value of an event in another stream k positions behind, using the construct $s[-k, x]$. If $-k$ corresponds to an offset beyond the start of the trace, value x is used instead. For example, the stream $s_1 = t_1[+1, false]$ which is obtained by taking at each position i the value corresponding to another stream t_1 at position $i + 1$, except at the last position, which assumes the default value false. Moreover, the language provides the expression $\text{ite}(b; s_1; s_2)$, which represents an if-then-else construct: the value returned depends on whether the predicate of the first operand evaluates to true.

LOLA can be used to model RV as a stream computation. Consider the specification property "every red light should not be followed by a yellow light"; suppose that g, r and y are two input streams of Boolean events, representing green, red and yellow light events respectively. Using LOLA, the property could be expressed as follows:

$$t := y[1, false]$$
$$\varphi := \neg(r \wedge t)$$

The equation t checks if the next event is yellow, except at the last position, which assumes the default value false. The equation φ returns False whenever $\neg(r \wedge \neg t)$ is True, i.e. whenever a red light appears and a yellow light appears in the next position (t evaluates to False), and True otherwise. This output can be used as the monitor verdict for the property.

As seen, the stream RV (SRV) as pioneered by LOLA is specialized for specifying synchronous streams, which means that events arrive in discrete steps where every input stream has an event at every step and all output streams produce an event. This is suitable for monitoring of correctness properties and performing quantitative measures. However, it is not appropriate for processing events that arrive at different frequencies and have arbitrary real-time timestamps, such as in cyber–physical systems, where timing is a critical issue.

2.3.6. Tessla: Temporal stream-based specification language

TeSSLa [84] is an asynchronous specification language that natively supports timestamped events. It mandates a global order for all stream events, but it does not necessitate all streams to have events occurring simultaneously. This enables modeling high-frequency streams.

An event stream in TeSSLa can be specified over a time domain $\mathbb{T}$ and a data domain $\mathbb{D}$ as a finite or infinite sequence $s = a_0t_0, a_1t_1 \ldots \in \mathbb{TD}$. To model a specification property, the language has many well-defined operators which can be used to transform an input stream of events into another stream. Given an input stream *write* that provides and write events to a file. The stream *write* can be in the form $write = wt_0, -t_1, wt_2, wt_3, -t_4, -t_5, wt_6$..., where w means that a write event happens and - means no event happens. The following specification checks whether the lapse of time between two write events exceeds 5 time units.

$$difference := time(write) - last(time(write), write)$$
$$output := filter(difference > 5, difference - 5)$$

The $time(write)$ operator accesses the timestamp each event in the $write$ stream. The *last* operator applies the operator $time(write)$ on the previous event. The stream *difference* computes the time difference between the current w event and the previous one. The stream $difference - 5$ is filtered by the condition $difference > 5$ using the *filter* operator. The resulting stream *output* is a sequence of output verdicts.

Note that TeSSLa is enriched with many other operators, such as the *delay* operator, which can create events at certain points. For example, the above property can raise a unit event on the *output* stream as soon as we know that there was no write event:

$$timeout := const(5)(write)$$
$$output := delay(timeout, write)$$

The first equation maps the values of events to the constant value of 5, which is then used as timeout value. In other words, the *timeout* stream is derived from the $write$ stream by replacing

each w event with the constant 5. In the second equation, the *delay* function works as a timer, which is set to a timeout value with the first argument and reset with any w event on the second argument. After 5 consecutive timestamps without a w event, an error is raised in the *output* stream.

2.3.7. Other specification languages

In preceding sections, we provided an overview of the principal specification languages employed in the relevant literature concerning RV with incomplete traces. These will be cited in Sections 4 and 5. Nonetheless, numerous alternative specification language tools and RV frameworks are utilized in the literature. We provide a concise overview of them in this subsection.

We start with the Runtime Monitoring Language (RML) [85,86], a simple yet potent Domain Specific Language (DSL) specifically devised for RV. RML is entirely modular and detached from the instrumentation and the specific type of system under scrutiny. The foundation and interpretation of RML hinge on a fundamental calculus known as Trace Calculus (TC) that boasts significant expressiveness, accommodating operators like prefix, concatenation, union, intersection, shuffle, and recursion. Additionally, it is parametric, accommodating specifications dependent on runtime-discovered values, generic enough to make certain specification parts reusable through abstraction over variables, and capable of handling infinite traces.

Larva [87–89] is a Java-based runtime verification tool. It uses symbolic automata as the foundational structure for its specification language. This design allows users acquainted with finite state machines to seamlessly transition to specifying properties and ensures Turing completeness by allowing the incorporation of Java code within transitions. Another notable hallmark of Larva is its *foreach* construct, a mechanism that simplifies the inclusion of top-level universal quantification within specifications. The tool also features built-in timers that can either trigger or safeguard transitions, streamlining the establishment of real-time properties. Additionally, for the facilitation of modular property definitions, Larva permits communication between properties through non-blocking channels, enabling the exchange of Java objects across monitors via internal communication events.

Other tools share similarities with Larva in terms of architecture and purpose, most notably JavaMOP [90] and MarQ [91]. The distinctive contribution of the Larva-associated body of work lies in its divergence from conventional specification languages, particularly LTL, in favor of an automaton-based notation.

Hawk [46] is programming-oriented extension of the rule-based Eagle logic. On its side, Eagle [92,93] is a runtime verification tool that encompasses both a rule-based language and an accompanying interpreter. This comprehensive framework supports an array of temporal logics, including future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints, as well as statistical analysis. Each transition within Eagle carries a condition, not only pertaining to the state machine input but also involving the variables constituting the underlying system. These transitions also incorporate an action that influences the state variables. Notably, the rules formulated in the Eagle system can embody either maximal or minimal fixpoint semantics, providing the flexibility to articulate both weak and strong interpretations of identical operators.

Drawing inspiration from PSLang, ConSpec [94] is a specification language tailored to operate within the confines of resource-constrained mobile devices. The framework mandates the formulation of a distinct contract for every application. Subsequent to installation on a device, this contract undergoes rigorous scrutiny against the user's specified policies. In scenarios where the stipulated contract fails to align with the user's policies, the application finds itself barred from installation onto the device. Alternatively, for instances where a comprehensive pre-installation contract evaluation remains unfeasible, a runtime monitor is seamlessly integrated into the application, allowing ongoing assessment post-installation.

3. Incomplete and uncertain sources of events

As we have seen in the previous section, existing approaches to RV make use of a large number of models for the representation of events, as well as the expression of properties. There are almost as many tools and models as there are combinations of traces and specification languages, and some works have even attempted to define conversions to go from one to another [30]. However, the vast majority of these approaches are underpinned by a fundamental assumption: the trace on which the monitoring is carried out is complete, and all the events it contains are exact and devoid of any error or uncertainty. Regardless of the condition to be evaluated and the notation used to represent it, the verdict produced by a monitor is reliable only if this crucial condition is respected.

Yet, one can easily imagine situations where the contents of a trace may not entirely be trusted: events may go missing, numerical measurements may carry an intrinsic uncertainty, etc. We shall group under the term "data restriction" any situation where an input trace is considered unreliable, regardless of the reason. As we shall see in Section 4, some works in the field of RV address the issue in different ways. However, before even describing how the problem can be tackled, it is appropriate to discuss the various ways in which an input trace can become incomplete or uncertain. In this section, we present a synthesis of the various causes for such partial information that have been invoked in the literature.

3.1. Mechanisms of data restriction

A first element that needs to be studied is the actual location in the monitoring process where data restriction takes place, and in what way this restriction affects the evaluation of a property on a trace. Fig. 7 represents a general view of the situations where data restriction may happen. In this figure, D is the original or "perfect" version of a data object (i.e. the input trace), while D' is a degraded, modified, or otherwise "unreliable" version of D.

A monitor M can be viewed as a process that performs a read operation on the contents of the data object, which can be likened to a form of "query" Q. The result of this query R corresponds to the trace (or part of the trace) whose content is needed by the monitor. For example, one could view the access to each individual event of a trace as a form of query-response loop that the monitor needs to perform in order to evaluate a given property. The figure represents four situations that can occur with respect to data restrictions. Situation 0, on the left-hand side of the figure, corresponds to the case where no data restriction occurs. Monitor M_0 performs a read operation Q_0 on the contents of the data object D and obtains the exact value in response R_0.

In situation 1, on the right-hand side, the monitor does not access the original data object D, but rather its restricted version D'. The monitor can still freely query the restricted data object D' by sending the query Q_1 and receiving a response R_1. This situation is representative of cases of (unintentional) data corruption, but also of deliberate restrictions meant to prevent access to the original trace contents. For example, values in a data object may be subject to anonymization, or parts of the object may simply be deleted to avoid unauthorized access.

In situation 2, at the bottom of the figure, the monitor M_2 queries the data object, but the original query Q_2 is transformed into a less precise query Q_2' – or blocked altogether. The monitor will receive the "correct" response R_2, but for the modified query

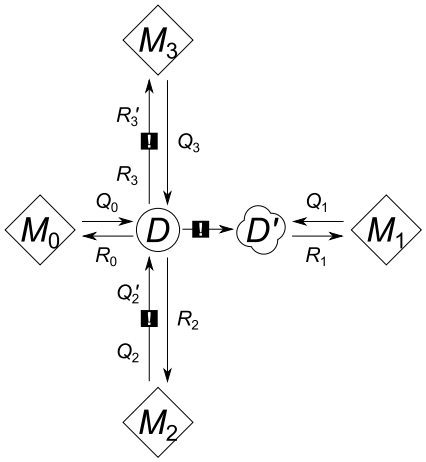

Fig. 7. An overview of data restriction situations.

Q_2' which probably queries different or less precise information than Q_2. Access control policies can be a reason behind blocking the query in this situation. Finally, in situation 3, the roles of query and response are reversed. The monitor can query whatever it wants, but the response may get transformed before reaching it. This situation is similar to situation 1, however the modifications in this situation are applied to the output of the query, and not to the data object itself.

All three situations can have the same observable effect to the monitor: receiving imprecise data, or even nothing at all. However, the difference lies in the mechanism by which uncertainty or imprecision is introduced.

3.2. Causes of data restrictions

Independent of the mechanism by which data restriction occurs, causes of incomplete data in traces can be broadly divided into intentional and unintentional causes. Intentional causes are the restriction mechanisms enforced by the user; hence they are expected, such as data restricted due to an access control policy. On the other hand, unintentional causes are unexpected phenomena that cause a loss of data, such as a sudden data corruption.

In a runtime monitoring context, both the intentional and non-intentional causes will affect the monitoring process due to their impact on the quantity and quality of the data available to monitor. In other words, some level of uncertainty will be introduced into the data fed to monitor, which differs based on the method of restriction imposed.

3.2.1. Intentional causes

We call "intentional causes" any deliberate operation that results in a degradation of the original input trace that can have an impact on the verdict returned by a monitor. For example, most information systems are equipped with mechanisms to prevent the disclosure of their confidential data. It can be an access control mechanism that determines who can access what, or one of several data protection techniques such as data, encryption, and data anonymization. In both cases, access to the data is restricted, and some query responses will be returned to the sender carrying incorrect or incomplete data or even no answer. We detail in the following some of the possible intentional causes listed in the literature.

Intentional causes can be represented by any of the situations 1, 2 and 3 of Fig. 7. In situation 1, the intentional modification can be applied to data source D to obtain D'. The user then queries D' instead of D and get imprecise response containing data different from the original data in D. Similarly, intentional modifications can be applied to Q_2 (resp. R_3) in situation 2 (resp. 3) to obtain Q_2' (resp. R_3'); the user will obtain the result of Q_2' (resp. R_3') instead of Q_2 (resp. R_3).

Access-control policy. An access control policy is a rule that defines who is authorized to access which data and under what circumstances he can do so. Several access control models are commonly used in computer systems [95–108]. Each model has different way of enforcing the rules, however all of them have the same aim to restrict the access to certain data. Data encryption can also be seen as a mechanism to enforce access control as it hides the data by converting it from a readable format into an unreadable encoded format [109].

As a simple example, consider a log containing medical records, where the field "diagnosis result" of each patient can only be accessed by a doctor. A runtime monitor verifying the satisfaction of property such as "the number of patients diagnosed by cancer is equal to 60" would need to access the restricted data values to be able to compute the number of patients with cancer to produce a certain verdict. With no access to such data, the monitor will produce an inconclusive verdict.

The situation of access control policy is represented in situation 3 in Fig. 7: if Q_3 requests to access objects O_1 and O_2 while an access control policy states that the requester is permitted only to access O_1, instead of receiving R_3, the requester will get an incomplete or a reduced response R_3' where the object O_2 is missing.

Data anonymization. Data anonymization is a technique used to protect sensitive data by hiding personally identifiable information while maintaining the integrity of the data [110]. The process of data anonymization introduces uncertainty into data that was initially certain.

There are several data anonymization techniques [111]. *Generalization* consists of reducing the precision of attribute values by changing their scale. For example, a discrete numerical data (such as age) value can be replaced by an interval of values where one of these values is the correct original value (such as $[30 - 40]$), and a categorical data value (such as city name) in the original data set can be replaced by a set of possible data values (such as {Montréal, Laval, Longueuil}). Each value in the data set or data interval is considered as one possible "world". A monitor accessing certain data values from the anonymized data set will get a set of possible events instead of one precise event.

Suppression is another anonymization technique, which this time completely deletes a data attribute or a part of the data set. On its side, *replacement* is a method that consists of substituting of characters of an attribute or value of the data with a predefined symbol (such as X or *). Note that, in case of complete suppression of a data value, the monitor will get a missing event. In case of partially masking characters, the monitor will get partially incomplete events (an uncertain event where a part of it is missing). Another type of anonymization is the slight modification of data attributes by adding some (random) noise to make them less accurate. This could be, for example, adding or subtracting days or months to a date. In this case the monitor will get incorrect or corrupted events.

Data perturbation. While data anonymization techniques only seek to completely remove or mask identifying information from the data set to prevent linking the data back to specific individuals or entities, data perturbation techniques seek to balance the need for privacy protection with the need to maintain the usefulness and accuracy of the data for analysis and decision-making [112,113]. There are various techniques for data perturbation, including randomization and noise addition, and data swapping.

One method of randomization is called projection perturbation [114], which is a geometric data perturbation technique applied to a data set whose values are represented as data points in a multidimensional space. A set of data points is projected from the original multidimensional space to another randomly chosen space. Another perturbation method is by noise addition [115–117] where certain amount of random noise can be added while the specific information, such as the column distribution, can still be effectively reconstructed from the perturbed data. For example, in the case of an execution log, suppose that a sequence of events is made of successive numerical values $x_1, x_2, \ldots, x_n$. A possible way of applying data perturbation would be to change the original data by adding random noise values $\bar{r} = r_1, \ldots, r_n$ to the original data, thus producing a modified trace $x_1 + r_1, x_2 + r_2, \ldots, x_n + r_n$, which would be published along with the distribution of $\bar{r}$. Such a perturbation makes it impossible to recover the original content of each event, but still preserves the validity of coarse-grained properties applying on the set of values. For example, a property expressing a condition on the average of the events is likely to produce the same verdict on the original and the modified trace.

With respect to data perturbation by swapping, it is applied by exchanging or swapping data values. In runtime monitoring, it could be by swapping data values of an event attributes of an event or swapping entire events in a trace. For example, in a traffic light data set, if the green light event is swapped with the yellow light event and a monitor is checking the property *a red light should be always followed by a green light*, the resulting verdicts will be imprecise due to the swapped events.

Load shedding. Load shedding is a technique for removing extra load from a system so that the overhead is reduced and the processing time keeps up with the rate of input arrivals when they become overloaded [118]. However, due of load shedding, the resulting data sets after load shedding can have varying levels of accuracy as a result of missing data values [119,120].

Load shedding is mostly applied in the data stream management systems, where the processing delay is the most important quality metric. In cases of overloading, which are typical in data stream systems, the ability to maintain a desired level of delay is severely limited. Joshi et al. [5] describe a scenario where a media player software is instrumented with a library given a fixed time budget. In a given time interval T, the instrumentation can produce at most B events; any event exceeding this threshold within this interval is replaced by a special "non-event" called χ.

Data sampling. Is a technique employed to systematically select a subset of data values from a pre-defined population to serve as a data source for data analysis tools and RV monitors [121]. Sampling techniques are broadly divided into two categories: probability and non-probability sampling. With probability sampling, one can specify the probability of an element (such as the events having the attribute x equal to a certain value n) being included in the sample. Among the probability sampling techniques, we have the "simple random sampling", where each element has an equal chance of being selected, and "stratified random sampling" where each element has a known probability of being selected.

In the case of non-probability sampling, the probability of including an element in the sample cannot be estimated. Hence it is less expensive than the probability sampling. Among the non-probability sampling, we have "quota sampling" where quotas are set for the number of elements to be included in the sample based on certain characteristics. These quotas are determined based on a prior knowledge of the population, and the "convenience sampling" which involves selecting sample units based on their accessibility to the selector, which could be influenced by various factors such as geographic proximity, availability during the study period, or willingness to participate in the analysis.

It is known that the monitor reports only the result of processing the events that it observes. Hence, a precise monitoring result depends on a precise sampling technique. In other words, by selecting a representative subset of the trace for analysis, RV process can be made more efficient, while still providing a high degree of confidence in the correctness of the system's behavior. Data sampling has been used to mitigate the computational overhead in runtime monitoring [122]: Arnold et al. [123] presented a runtime environment that can efficiently check the violations of user-specified correctness properties with a controlled overhead. They introduced property-guided sampling and in particular object-centric sampling, to collect sampled profiles while preserving correctness of the analysis. Property-guided sampling ensures that the sampled profile maintains sufficient properties to make the dynamic analysis meaningful. Object-centric sampling allows an analysis to sample at the object instance level. An object can be marked as tracked, and the analysis can receive all profile events for this object while receiving no events for untracked objects.

Other monitoring approaches tend to do sampling by temporarily disabling monitoring process; this is the case of Huang et al. [124], whose proposed technique temporarily disables monitoring of selected events for the shortest possible duration while ensuring that the user-specified target overhead is not exceeded. Fei et al.'s [125] method selectively enables monitoring for specific function executions. By default, their method tracks a function execution only if it is called in a previously unseen context. Theoretically, a function's context encompasses all memory locations it accesses. Storing and comparing all such contexts would be prohibitively costly. They use less demanding definitions of "context" and "context matching" which may lead to missing some interesting behaviors.

3.2.2. Non-intentional causes

Apart from the intentional causes enumerated above, there also exist unforeseen situations that result in data restriction. All non-intentional causes belong to situation 1 of Fig. 7 where the data source D is changed to D' after applying certain modification technique(s). The user will query D' instead of D and get imprecise response containing data different from the original data existing in D.

Data corruption. A first obvious non-intentional cause is data corruption. Events in a trace can be stored on a medium that degrades over time and may render access to some of their values impossible; error detection codes, such as CRC-32, can also reveal that stored data is invalid, without necessarily providing the means to recover the original data. In such a situation, all one can know is that an event occurred or that some value was recorded, but that the actual contents cannot be trusted.

Another common type of data corruption occurs during data transmission, when a data event or an interval of events is dropped from the stream, for example, due to a momentary communication link failure or as a result of environmental factors interfering with data transmission, particularly when using wireless transmission methods. Assuming that each transmitted data value is assigned a unique and incrementing ID, the presence of non-successive IDs can be used by a user connected to the source where these events are stored to detect the occurrence of such a drop. This makes it possible to determine how many events occurred, but not their values.

Incorrect system instrumentation. As mentioned in Section 2.1.2, the instrumentation is a computational process that extracts and records events from a software system during execution to make them available for analysis by a decision procedure (such as an RV monitor) [39]. The recorded events are sent to the monitor as an ordered stream (a trace of events). The event order in the execution trace is usually guaranteed by instrumentation to correspond to the order in which the appropriate computing step occurred. However, in some cases, such as distributed environments, only a partial ordering of events can be properly relayed to the monitor.

There are other situations where logging statements are manually inserted by the developers [17,18]. In such a context, many relevant logging statements can be missing from a system [126]. Each logging statement typically is assigned a log level. There are typically six types of log levels ordered based on their verbosity: `TRACE` > `DEBUG` > `INFO` > `WARN` > `ERROR` > `FATAL`. The usage of these levels by developers can be highly unreliable [127] where the same statement in two distinct code locations can be assigned two different levels (e.g. `INFO` vs. `DEBUG`). For example, if a user sets the verbosity level to be printed at the `WARN` level, only the logging statements with the level `WARN` or `ERROR` or `FATAL` would be printed out (and thus reach a monitor). If a relevant event for the evaluation of a property is assigned the incorrect level, it runs the risk of being filtered out on the grounds of the verbosity level and not reach the monitor.

Such manually-generated logging statements can also be imprecise in themselves. For example, suppose that a message such as "Error reading resource" can be used to indicate either a disk or a network failure; a monitor for a property such as "every disk failure must stop the program" may report incorrect violations because two types of failure get the same message and translate into the same event.

Imprecise measurements. As was discussed in Section 2, in some situations events contain values that have been measured by some sensor device. These devices might suffer from low data quality due to long-term use and other environmental factors [128] resulting in bias, drifting, full failure, or precision loss and other faults in the data recording process. Supplying a decision procedure such that a runtime monitor with inaccurate data from sensors will affect the verdict produced by the monitor. A simple example from Taleb et al. [2] considers a sensor recording temperature producing a value T having an error range, e.g. $T = 20^\circ \pm 0.5$. If the verdict produced when monitoring a property depends on whether $T \leq 20$ or $T > 20$, the monitor will not be able to produce a definite verdict for a range of values of T.

Another example of such a situation is illustrated by the monitoring of the position of a drone [129]. The altitude a of the drone can be modeled as a probability distribution. In such a model, a Boolean statement such as $a > 3$ cannot be expressed directly, as the precise value of a is unknown. One can only speak of the probability $Pr(a > 3)$; in such properties, Boolean statements are recovered by giving bounds, such as $Pr(a > 3) \geq 0.99$.

Impedance mismatch. Impedance mismatch is a cause of data uncertainty that occurs while checking a property over a trace of events during RV. In order to monitor an event, the property is checked over the event parameters and emits a verdict if the parameters of the event are compatible with the property. Usually, we can solve this issue by rewriting the property so that it can align with the instrumentation and the event parameters. Impedance mismatch occurs if two conditions are satisfied: first, there is no knowledge about event parameters. Second, the parameters used to express the property do not align with the event parameters and there is no possibility to re-write the property so that it matches the event parameters.

For example, a property may specify conditions on individual values of x and y, while the source of events only gives their sum s. Taleb et al. suggest that impedance mismatch can occur, for instance, when one wishes to monitor a new property over a log that has been recorded for another purpose [2]. One solution that does not require rewriting the property is to turn values of s into imprecise versions of x and y.

Decentralized and distributed systems. A distributed and decentralized monitoring setting is typically built from subsystems or processes situated at various locations or nodes [130]. These entities operate independently and establish communication among themselves through an underlying communication platform. Each location retains a record of events that occur. In a runtime monitoring configuration, monitor instances are also dispersed across different locations and possess the capability to communicate with one another. This arrangement offers the flexibility to conduct property checks in a decentralized manner.

In real-world scenarios, both non-distributed and distributed systems are susceptible to encountering failures. Nevertheless, failures within distributed systems can be more intricate compared to non-distributed systems due to the separate nature of the executing units. Communication between nodes can introduce loss of information; nodes can fail due to hardware issues or software bugs, or may behave maliciously or provide incorrect information. These failures are known as Byzantine failures and can lead to data corruption and uncertainty about the integrity of the system. Systems that use replication for fault tolerance, it can be challenging to ensure consistency across replicated copies of data. If updates to data are not synchronized correctly, data inconsistencies may occur, leading to uncertainty about which version of the data is accurate.

3.2.3. Effects of data restrictions

In the preceding sections, we described the mechanisms of data restrictions and all the causes of data restrictions that can happen intentionally and non-intentionally. We hinted by means of a few examples to the impact that these restrictions can have on the verdict produced by a monitor. In this section, we examine this notion in more detail and discuss the possible effects of data restrictions on the monitoring process.

Consider a simple situation where possible atomic events are $\Sigma = \{a, b, c\}$, a trace $\overline{\sigma} = abcababbca$, and the simple property that stipulates that "every a must immediately be followed by b". If the monitor is fed event a, and the subsequent event b is dropped from the trace, it will incorrectly conclude that the property is violated upon receiving the next event c. The same will happen if, instead of being dropped, b is corrupted and turned into event c. In those situations, the monitor reaches a definitive verdict, but this verdict is incorrect in light of the content of the original trace.

A different set of issues can arise if the presence or content of events is uncertain. For example, suppose that the actual identity of the second event of the trace is not known. In such a situation, the monitor cannot reach a definitive verdict: the property could be satisfied (if the unknown event is b) or violated (if it is anything else). A similar outcome occurs in the situation where b may or may not have occurred.

We distinguish between eight types of data restrictions. This categorization will be used in later sections to classify the works on RV under uncertainty according to the type of restriction they consider.

Case 1: We know exactly one event is missing and where. In this first case, the monitor is given a trace where the number and location of missing events is known. For example, a monitor might receive the input trace $\overline{\sigma} = abc\chi babbca$, where χ is marker indicating that at this precise location, an event is known to have occurred but was lost. We have seen this happens in some cases of load shedding where actual events are dropped and replaced by an empty "non-event". This can also occur in situations where each event is given a sequential number, and where a gap in the order of these numbers is detected.

In such a situation, the monitor may ignore the missing events and proceed with the next event, or generate a non-conclusive verdict, or consider the set of all possible events that may occur in this gap. Note that this case can be extended to the situation where n successive events are known to be lost (which would be detected by the presence of multiple successive χ markers), or multiple individual events are missing throughout the input trace.

Case 2: We know an event is invalid, but we can't recover its contents. This case is handled in the same way as the previous one. A corrupted event can be considered as a missing event whose occurrence is known. As we discussed earlier, corruption can be made known by means of checksums and other integrity checks, which can typically uncover the presence of corruption but not always recover from it.

Case 3: We know events are missing, but we do not know how many. This time the χ marker may only be interpreted as the presence of an *interval* of missing events, but the number of events in this interval is unknown. Thus a runtime monitor may receive the trace $\overline{\sigma} = abc?abbca$, where this time ? indicates the location of an interval of missing events. This could occur, for example, when the communication link feeding events to the monitor is interrupted and then resumed, but without the presence of sequential numbers that could indicate how many events have been lost in the meantime.

This case is much harder to handle than the previous two, due to the higher degree of uncertainty on the contents of the trace. Yet, in some cases, a monitor can still recover from such situations and produce a sound verdict. For example, if the monitor evaluates the property "every c is eventually followed by an a", it could conclude that the received prefix satisfies the property regardless of the length and content of the missing gap.

Case 4: Events are missing and we don't know about it. In this case, no marker is even present to signal possibly missing events. Thus, a monitor would receive for example the trace $\overline{\sigma} = abcabbca$; the monitor is not notified of whether, if any, and where, are missing events in this input trace. As with case 3, a monitor could still produce a valid verdict for some input traces and some properties, however it does not even have a mean of knowing when its verdict could be incorrect. We list this situation for the sake of completion, but it goes without saying that none of the surveyed works address this situation.

Case 5: Events are corrupted and we don't know about it. This is equivalent to case 4. The monitor will process the event as if ignoring the presence of corrupted events.

Case 6: We know an event may be one from a set, but we don't know which one. This can be seen as a more precise type of uncertainty than cases 1 and 2. Instead of supposing that a missing event could be any one in Σ, this time the monitor is given slightly more precise information as a set of possible events is known. One solution could be replacing the event with a set of possible replacements or by the conjunction of the elements of this set. For example, the monitor could receive the trace $\overline{\sigma} = abc\{b, c\}babbca$, where the exact value of the fourth event is unknown, but it can only be b or c.

This happens, for example, if an event is partially corrupted, so that its contents is known in part (enough to eliminate a set of possibilities over what it could be). It is also a symbolic way of representing uncertainty over numerical values; thus a value of 20 ± 0.5 indicates that the "true" value can be any one in the interval $[19.5, 20.5]$, without knowing exactly which one it is.

Case 7: We know an event x may or may not have occurred. In this situation, the monitor is fed events, but some of them have a marker indicating that their occurrence is uncertain. For example, a monitor could receive a trace $\overline{\sigma} = ab\dot{c}abbca$, where the dot over the first c indicates that this event may or may not have occurred. Conceptually, this case can be handled in a way similar to Case 6, if one allows the empty event ϵ to be one of the possibilities.

Case 8: We know events x and y occurred, but we don't know which came first. This situation happens in cases where the interleaving of multiple events is not precisely known, such as in the Life Data Recorder discussed earlier. In this case, the monitor could receive a trace such as $\overline{\sigma} = ab(c \parallel a)bbca$, where $c \parallel a$ indicates that both c and a have occurred, but their exact ordering is missing. The monitor in this case could consider the two possibilities $\{ca, ac\}$ and produce a set of two possible verdicts.

We shall mention that, depending on the specification property, the monitor may be able to produce a conclusive verdict regardless of what and how many the missing events are. For example, if the property states that each b should be finally followed by a, once receiving the event a after the gap, the monitor is able to produce a conclusive and sound verdict. Moreover, in some situations and for some specification properties, extending an existing specification language with useful operators allows writing a specification property in a way that avoids the need for the missing or uncertain event in producing a correct verdict [6].

4. RV approaches to data restrictions

As explained in Section 3, the presence of data restrictions can be caused by a variety of factors, either intentional or unintentional. Moreover, data restrictions obviously have an impact on the verdict produced by a monitor in some situations. In this section, we survey and categorize the various approaches that have been taken in RV literature to address this issue.

Two types of data restrictions must be distinguished at the onset: *unknown* restrictions correspond to the first example, where the monitor has no means of assessing where and how data restriction occurs; for example, when an event is dropped and no mechanism exists to inform the monitor of its absence, or when its contents are corrupted and it is impossible to discover this. This type of data restriction commonly occurs in distributed systems where identifying the root cause of data loss or uncertainty can be complex.

By definition, it is impossible to *always* recover from *unknown* data restriction: the monitor will necessarily produce an incorrect verdict in some situations. For example, in distributed systems, monitoring and debugging require specialized tools and techniques, adding uncertainty to the troubleshooting process. However, some approaches consider RV for distributed systems by keeping into account this type of restrictions by relying on the programming language or model used to implement the monitors. Audrito et al. [131,132] for example, define the behavior of the distributed devices by perceiving them as a collective computational entity that collectively executes a distributed computational process, utilizing the field calculus as a fundamental programming language. Aggregate computing is decentralized by

nature. The absence of peer-to-peer connections and an abstraction from actual message transmission leads to a fault-tolerant achieved by construction, as missing or delayed messages or network partitioning are indistinguishable from normal operation. Moreover, relying on the field calculus, their approach can self-adapt to changes in the network topology. Thus, it inherently manages instances of failure without manual intervention.

The rest of the works surveyed in this section address *known* data restrictions. These correspond to alterations of the input trace that the monitor is made aware of. Examples of this type of uncertainty include: a numerical measurement accompanied by an interval of uncertainty; a placeholder indicating that an event occurred without knowledge of its actual contents; or a mechanism that can identify that a data object is corrupted without the capability of recovering its contents. In those cases, a monitor can warn its user that the presence of data restrictions may have an impact on the accuracy or the validity of its verdict.

The nature of this warning varies from one study to another: some works propose a verdict associated to a probability; other works output multiple possible verdicts. As we shall see, some approaches use statistical methods to create a RV model capable of computing a final verdict, while others build the model using formal languages and automata theory, and many approaches work on the abstraction of incomplete event traces to achieve an abstract verdict.

In this section, we describe approaches from literature that tackle the problem of RV with incomplete or imprecise data and we classify them into abstraction based approaches, statistical-based approaches and language-based approaches. Each of these approach is described in the same way, by summarizing the following elements:

- The type of uncertainty targeted, by linking them to the various cases enumerated in Section 3.2.3
- The type of events (atomic, numerical, tuples, etc.) and the way uncertainty about them is represented
- The method used to represent the specification and the type of verdict produced by the monitor (e.g. probability, interval, set of possible values, etc.)

The formalism used by each approach to represent the events and the specification properties are listed in Table 4.

Furthermore, we divide the existing works into three broad families of techniques: abstraction-based approaches (Section 4.1), language-based approaches (Section 4.2), and statistical-based approaches (Section 4.3).

4.1. Abstraction-based solutions

Some RV approaches use abstraction methods to solve the problem of monitoring a property over an incomplete trace. Attempting to fill a gap in a trace with all possible replacements for the missing or uncertain event will produce a large number of concrete traces. Abstracting the set of concrete traces into one abstract trace will simplify the approach. In this section, we discuss the RV approaches based on abstraction.

4.1.1. Taleb et al. [2]: RV under access restrictions

Type of uncertainty targeted:. First, Taleb et al. proposed a logical framework that accounts for incomplete and uncertain information due to several reasons that restrict a monitor's access to the source of events. A proxy is used to model different kinds of access restrictions including missing events, corrupted events, encrypted events, partially unknown events, and correlated uncertainty where the deterioration is correlated with other events.

Type of events and their representation. An event is a valuation that assigns to each atomic proposition a truth value from domain $\mathbb{B} = \{\bot, \top\}$. For example, if $A = \{a, b, c\}$ is a set of atomic propositions, an event is represented as $e = \{a = \bot, b = \top, c = \top\}$. This is also called a uni-event because it contains one valuation or one possible world only. The proxy is built using a finite-state machine. It takes a valuation as input and models the uncertainty by generating all the possible replacements (this represents cases 6 and 7 of Section 3.2.3). The output is a set of valuations called a multi-event. For example, one kind of uncertainty is: *it is impossible to know which one of the propositions a or b holds in an input event, only that at least one of them is true*. In this case the proxy will replace an input event that supports a by an output multi-event that only supports the weaker proposition $a \vee b$. In other words, the event $e = \{a = \top, b = \bot, c = \bot\}$ where a holds is transformed to the multi-event $v = \{\{a = \top, b = \bot, c = \bot\}, \{a = \bot, b = \top, c = \bot\}, \{a = \top, b = \top, c = \bot\}\}$, where at least a or b holds. Similarly for events where b holds. A trace of multi-events is called a *multi-trace*.

Method used to represent specification property and type of verdict. With respect to the specification property, Taleb et al. use a mealy machine to lift a multi-monitor (that can accept a multi-event as input) from a uni-monitor (that can accept a uni-event). For a given multi-trace, the output of the multi-monitor is the set of outputs obtained by running the underlying uni-monitor on every possible uni-projection (every possible path or sequence of uni-events producing an output verdict). This set of outputs is called a multi-verdict. On the other hand, the fact that events fed to a monitor can now contain multiple valuations has an impact on the possible verdicts produced by the monitor where two uni-projections may result in two different verdicts. This ambiguity can be measured by associating each verdict to the fraction of all uni-traces that yield this verdict and hence can be used as a quantitative indication of its likelihood.

4.1.2. Leucker et al. [3]: RV for timed event streams with partial information

Type of uncertainty targeted. For their part, Leucker et al. proposed a solution for RV over streams of data containing missing and imprecise values. A data stream is a sequence of timestamps and data values representing the stream's events. To model imprecise values, streams are lifted from concrete domains of data to abstract domains. For example, a concrete numerical value in a concrete stream can be represented as an interval of real numbers in the abstract stream. Briefly, an abstract event stream is represented as multiple concrete event streams carrying information about the events and the gaps (this represents cases 6 and 7 of Section 3.2.3).

Type of events and their representation. With respect to event representation, a concrete event at a timestamp t can be a known event d of any type (such as Boolean) belonging to a data domain $\mathbb{D}$, $\bot$ if there is no event at t, or ? for timestamps after the progress of the stream. A data abstraction of a data domain $\mathbb{D}$ is an abstract domain $\mathbb{D}^{\#}$ where a particular point t can either be a known event from $\mathbb{D}$ with a known timestamp, $\bot$ if there is no event at t (but there are events at $t' > t$), $\top$ if there is an event at t but it is unknown (imprecise), and $\smile$ to represent a gap (a segment of an abstract event stream that represents all combinations of events that could possibly occur in that segment, both in terms of timestamps and values).

Method used to represent specification property and the verdict type. Leucker et al. extended the TeSSLa specification language described in Section 2.3.6 into *Abstract TeSSLa* by defining an abstract counterpart operator for each concrete operator of TeSSLa.

This allows deriving an abstract specification property from a concrete specification property by replacing every concrete TeSSLa operator with its abstract counterpart.

The abstract specification is proved to be a sound abstraction of the concrete specification, i.e., every concrete verdict generated by the original specification on a set S of possible input traces is represented by the abstract verdict applied to an abstraction of S. For example, in the domain $\mathbb{B}$, a concrete event can be *true*, *false*, or $\bot$. Applying a concrete TeSSLa specification, we get a conclusive concrete verdict for the *true* and *false* events, and a non-conclusive verdict when encountering $\bot$ (missing event) or an imprecise event or a gap of any length. However, for an abstract trace in the domain $\mathbb{B}^{\#}$, an abstract verdict (set of concrete verdicts) is produced when applying the abstract specification over the abstract events. For a known event, the resulting abstract verdict contains one concrete conclusive verdict. For a missing event, which is still represented as $\bot$, the abstract verdict is the same as the verdict produced on a concrete event $\bot$. For an imprecise event replaced by $\top$ which represents any possible event from $\mathbb{B}$, the abstract verdict is a set containing all the possible conclusive verdicts. For a gap replaced by $\smile$, the abstract verdict is a set of all possible conclusive and non-conclusive verdicts.

4.1.3. Wang et al. [4]: RV of traces under recording uncertainty

Type of uncertainty targeted. Wang et al. [4] present an approach for RV to handle the uncertainty that arises due to imprecise order of events in a trace. A Life Data Recorder device (LDR) is used to collect updates to data variables such as x and y and stores their values as a snapshot vector (Section 2.2.5) or a frame in the memory. Some variables are process variables that are updated once in a frame, while other variables can be updated several times in the same frame. Uncertainty arises when the update of one variable interleaves with the other variables in the same frame and the knowledge about the exact ordering of their updates in the frame is lost. As a simple example, suppose that a process variable x is updated one time (from value 2 to value 3) in the frame f, and the variable y is updated two times (from value 4 to 3 and from 3 to 5) in the same frame f. One possible ordering of (x, y) updates could be $(2, 4) \xrightarrow{x} (3, 4) \xrightarrow{y} (3, 3) \xrightarrow{y} (3, 5)$. When x and y interleave, we cannot determine the exact ordering of (x, y) updates (this represents case 8 of Section 3.2.3).

Type of events and their representation. Wang et al. consider each frame recorded by LDR as an abstract state, and each mapping from the variables x and y to their values a concrete state. A possible concrete state is (2,4) which maps x to value 2 and y to value 4. Several traces of concrete states can be extracted from one abstract state such as:

$$(2, 4) \xrightarrow{x} (3, 4) \xrightarrow{y} (3, 3) \xrightarrow{y} (3, 5)$$

$$(2, 4) \xrightarrow{y} (2, 3) \xrightarrow{x} (3, 3) \xrightarrow{y} (3, 5)$$

$$(2, 4) \xrightarrow{y} (2, 3) \xrightarrow{y} (2, 5) \xrightarrow{x} (3, 5)$$

A sequence of abstract states form an abstract trace Tr. The set of concrete traces consistent with the abstract state $Tr(i)$ is represented as $Path(Tr(i))$.

Method used to represent specification property and the verdict type. Past LTL (Section 2.3.3) [1,26,133] is used to represent the monitoring property using atomic formulas such as $\odot\varphi$ (meaning that φ was true at the immediately previous state), $\diamond\varphi$ (meaning that there was some time in the past when φ was true), $\boxdot\varphi$ (meaning that φ was always true in the past), and $\phi S \psi$ (meaning that either ϕ was always true in the past, or ψ held somewhere in the past and since then ϕ has always been true).

Wang et al. aim to monitor a property over the abstract trace provided by the LDR. They keep the syntax for past-LTL and introduce a new three-valued semantics based on standard semantics for concrete traces. A formula φ evaluates to true ($\top$) on an abstract trace Tr only if it evaluates to $\top$ on all concrete traces consistent with Tr; it evaluates to false ($\bot$) on Tr only if it is $\bot$ on every concrete trace consistent with Tr; otherwise a non-conclusive ? is resulted. To monitor the property $\varphi = \phi S \psi$ over a concrete trace $p_0, \ldots, p_m$ a checking algorithm iterates through all concrete states from p_0 through p_m. In each concrete state p_j, the checker keeps the resulting verdicts of all subformulas (ϕ and ψ) on the trace $p_0, \ldots, p_{i-1}$ (called the checker state). The checker updates its state based on the values in p_i.

To monitor the formula $\varphi = \phi \mathbf{S} \psi$ over an abstract trace Tr, the semantics are built in a recursive fashion assuming the resulting verdicts of checking the subformulas ϕ and ψ over the partial trace $Tr(i)$ is finished and available in a mapping SV_i : $SubFormulas(\varphi) \rightarrow \{\top, \bot, ?\}$. The function $checkOne(SV_i; p; \varphi)$ is then used, where p is one concrete trace from $Path(Tr(i + 1))$, the function returns whether φ is satisfied on all, none, or some (neither all nor none) concrete traces formed by concatenating any concrete trace in $Path(Tr(i))$ with p.

Kallwies et al. [134] studied the problem of recurrent monitoring with partial knowledge about input events. Recurrent monitoring checks a property from a specific position t in the trace (not necessarily a prefix of the trace). Each event is represented as a tuple of atomic symbol and position in trace. If a violation of the property occurred, it is associated with this particular position t rather than the entire trace. Kallwies et al. extended recurrent monitoring to k-offset recurrent monitoring where the verdict that the monitor must compute is shifted by a constant offset k. They extend past-LTL with bounded future and propose anticipatory recurrent monitoring. The anticipatory monitor computes functions that predict the future verdicts of the original monitor which are possible after the current observation. It can be also used to handle uncertain events. An uncertain input event is modeled as a set of possible inputs that actually happen. Kallwies et al. also used assumptions to improve the anticipation. Another approach for Kallwies that deals with uncertainty using assumptions is in [135].

4.2. Using language-based solutions

Aside from statistical and abstraction-based methods, some approaches proposed a formal language equipped with useful operators to write a specification property that can produce conclusive verdicts when monitoring a trace with incomplete events.

4.2.1. Joshi et al. [5]: RV of LTL on lossy traces

Type of uncertainty targeted. They presented an approach to the problem of RV in the presence of transient loss, which is a non-permanent loss of an event or a finite sequence of events is lost in a trace. After the data loss, the number of events that happened is known but their content is unknown (this represents cases 1 and 2 of Section 3.2.3). The goal of the authors is to show that there are some properties that can be monitored regardless of the presence of lossy events, under the condition that the monitor is able to observe subsequent valid events after the loss.

Type of events and their representation. An event can be a single atomic proposition from an alphabet Σ or an atomic formula composed of atomic propositions connected using Boolean operators (such as conjunction $\vee$ and disjunction $\wedge$). A lossy event is represented by the symbol χ.

Method used to represent specification property and the verdict type. The specification property is expressed using LTL (Section 2.3.3) and converted into an RV-LTL monitor, which is a finite-state automaton presented by Bauer et al. [25] as an extension of the LTL$_3$ semantics into $\mathbb{B}_4 = \{\top, \top_p, \bot_p, \bot\}$, where $\top_p$ and $\bot_p$ are emitted whenever an observed system behavior has not yet lead to a violation or acceptance of the monitored property. The value $\top_p$ (respectively $\bot_p$) means that the system will *presumably* satisfy (respectively violate) the property in the future. In order to determine whether the property is monitorable over a lossy trace, Joshi et al. build an algorithm that searches for a loss-tolerant alphabet and a loss-tolerant cluster in the RV-LTL monitor. A loss-tolerant alphabet represents the input elements where each element forces the monitor to transition into a unique state irrespective of its current state. The monitor is supposed to move to the unique state at the end of the loss if the processed element after the loss belongs to the loss-tolerant alphabet. A loss-tolerant cluster constitutes the set of states where each state transits the monitor to the same next state within the cluster when processing the same input from the loss-tolerant alphabet. Hence, If a loss occurs when the current state of the monitor belongs to a loss-tolerant cluster, the transitions of the cluster ensure that the next state would still be one of the same cluster.

A loss-tolerant monitor $\mathcal{M}$ is derived from an RV-LTL monitor by adding a new state. Whenever a lossy element χ appears, the monitor moves to this state and outputs the verdict "?". Hence, a loss-tolerant monitor produces an output in the truth-domain $\mathbb{B}_5 = \{\top, \top_p, ?, \bot_p, \bot\}$ which is $\mathbb{B}_4$ augmented with "?".

4.2.2. Basin et al. [6]: Monitoring compliance policies over incomplete and disagreeing logs

Type of uncertainty targeted. They study the effect on RV of missing data due to logging failures and disagreement between logs about the occurrence of certain events when multiple logs are required to verify a property.

Type of events and their representation. Basin et al. represent the uncertainty over event occurrences by means of what they call a *logging knowledge base*. A knowledge base is a sequence $\overline{\mathcal{D}} = \mathcal{D}_0, \mathcal{D}_1, \ldots$ of first-order structures defined over the set of ternary Boolean values $\{\top, \bot, ?\}$, where "?" represents the unknown truth value. Each first-order structure represents a discrete time point, and totally defines the (ternary) truth value of each event predicate. Informally, for some predicate r of input arity n, $r(a_1, \ldots, a_n) = \top$ in a given time point τ indicates that the event $r(a_1, \ldots, a_n)$ with parameters $a_1, \ldots, a_n$ happened at τ. Conversely, $r(a_1, \ldots, a_n) = \bot$ in a given time point τ indicates that the event $r(a_1, \ldots, a_n)$ did not happen at τ. Finally, $r(a_1, \ldots, a_n) =?$ represents a *knowledge gap* with regard to whether $r(a_1, \ldots, a_n)$ happened at τ (this represents case 7 of Section 3.2.3).

Method used to represent specification property and the verdict type. Basin et al. propose what they call a compliance policy language L_3, which is a variant of First-Order Temporal Logic (FOTL) [136], to formalize and evaluate compliance policies in the presence of incomplete knowledge. A compliance policy is typically represented as a set of regulative normative statements (norms), that express what conditions need to be held by an agent to be authorized to do specific actions. Norms are applied at all times within a system, and deadlines are critical to manage temporal norms. Based on these notions, a compliance policy in L_3 is a closed formula of the form $\Box\forall\bar{x}.\varphi$.

For logging failure, Basin et al. assume that during the logging process, all events at are recorded correctly, and if a logging failure happens at a time point τ, the logging process stops and nothing is recorded until the process is restarted. Based on this assumption, policy violation could be avoided at τ (where the failure happens) if the policy to be checked depends on the past events that are already recorded before τ. The language L_3 is equipped with the temporal connective operator $\blacklozenge_{[b,b')}\varphi$ which returns *true* if φ is *true* at least at one past time point in the time interval $[max(0, \tau - b' - 1), \tau - b]$, and *false* if it is *false* at all the time points in this interval. For example, the compliance policy *If a request is serviced at a web-server, then it must not have been denied by a firewall (in the past x time points)* is formalized as $\Box\forall r.(\text{service}(r) \longrightarrow \neg\blacklozenge_{[0,x)}\text{deny}(r))$, where r is the request and service(r) and deny(r) are predicates respectively representing the servicing and denying events of the request r. If the failure happens at τ and we want to verify the predicate service(r) at τ, then all requests that had been denied at the previous x time points potentially violate the policy. However, if none of these time points has deny(r) hold, the policy is therefore satisfied. So, not all logging failures must result in potential violations.

L_3 also specifies the obligations that should be respected by two parties exchanging documents; for example, the policy "all received documents must be paid for within 5 days". L_3 provides the operator $\lozenge_{[0,6)}pay(d)$ which has the same interpretation as $\blacklozenge_{[b,b')}\varphi$ but for future time points, and the operator $\otimes$ which is better than $\vee$ and $\wedge$ in the sense that none of the parties will be favored over the other when they disagree about the occurrence of an event. The policy is stated in L_3 as follows: $\Box\forall d.send(d) \otimes receive(d) \longrightarrow \lozenge_{[0,6)}pay(d)$. If nothing is sent at τ, the receiver's log does not contain a receive(d) and the sender's log does not contain a send(d), then the receiver in this case will not pay anything ($false \otimes false \longrightarrow false$ evaluates to *true* meaning that the policy is satisfied). Contrarily, when a document is sent, we have that $true \otimes true \longrightarrow true$ evaluates to *true* meaning that the policy is also satisfied. However, the sender may insert fictitious send(d) events to oblige the receiver to pay while the receiver's log disagrees (no receive(d) event in the receiver's log). In this case, the $\otimes$ operator can be used: $true \otimes false$ evaluates to $\bot$, and $\bot \longrightarrow false$ evaluates to $\bot$. In this case, specification no longer favors one party over the other.

4.2.3. Basin et al. [7]: On real-time monitoring with imprecise timestamps

Type of uncertainty targeted. Basin et al. raised the problem of imprecise timestamps of traces influencing the correct verification of the properties.

Type of events and their representation. Two types of traces are considered: an observed trace and a real trace. The observed trace is a timed word $\overline{\sigma}$ containing imprecise timestamps and is represented as a sequence of tuples (τ_i, a_i) where $i \in \mathbb{N}$, $\tau_i \in \mathbb{T}$ ($\mathbb{T}$ is a discrete time domain) is the time stamp and $a_i \in 2^P$ is an atomic proposition from P. The real system trace, which contains precise timestamps, is represented as a timeline $\rho_{\overline{\sigma}}$.

To represent the imprecise timestamps, Basin et al. assume a timestamp imprecision $\delta \geq 0$, where an imprecise timestamp is assumed to belong to $[\tau_i - \delta, \tau_i + \delta]$ (this represents case 6 of Section 3.2.3 applied to timestamps instead of event values). A set of timelines $TL(\overline{\sigma})$ can be obtained from a timed word based on the function $\pi : \mathbb{T} \to 2^P$, where $\pi(t) = a_i$ if $ts^{-1} = \{i\}$ (where $ts : \mathbb{N} \to \mathbb{T}$ is an injective function and $ts(i) \in [\tau_i - \delta, \tau_i + \delta]$) or $\pi(t) = \emptyset$ otherwise). For example, if $\overline{\sigma} = (\{p\}, 1), (\{q\}, 1), (\{r\}, 2), (\{s\}, 5)...$ and $\delta = 1$, the time intervals are $[0, 2]$, $[0, 2]$, $[1, 3]$, $[4, 6]$ and a possible timeline is π where $\pi(0.6) = \{q\}, \pi(1.2) = \{r\}, \pi(1.3) = \{p\}$ and $\pi(t) = \emptyset$ for $t \in [0, 4)\backslash\{0.6, 1.2, 1.3\}$.

Method used to represent specification property and the verdict type. Basin et al. use MTL (Section 2.3.4) to rewrite φ into $tf(\varphi)$, where $tf(\varphi)$ accounts for timestamp imprecision by relaxing the implicit temporal constraints on atoms. For example, instead of having "p holds now", we have "p holds at a time point within the interval $[0, \delta]$ in the past starting from now or p will eventually hold at a time point within the interval $[0, \delta]$ from now". Formally, $p \in P$: $tf(p) := (\blacklozenge_{[0,\delta]} p) \vee (\lozenge_{[0,\delta]} p)$. On the other hand, they transform $\overline{\sigma}$ into a *monitored* timeline $\rho_{\overline{\sigma}}$ by ignoring timestamp imprecision. Then they use an existing monitor (for precisely timestamped traces) to monitor $\rho_{\overline{\sigma}}$ with respect to $tf(\varphi)$.

They aim to identify the MTL fragments φ for which conformance with $tf(\varphi)$ over $\rho_{\overline{\sigma}}$ implies conformance of all $\pi \in TL(\overline{\sigma})$ with φ, which consequently implies the satisfaction of φ over $\overline{\sigma}$. For example, if $\varphi = p$ and $tf(p)$ is satisfied at t, then p is satisfied at some t' within the interval $[t - \delta; t + \delta]$, and thus there is a possible timeline for which φ is satisfied at t. However, not all timelines satisfy φ at t. In this case, we cannot obtain guarantees about a precise verdict of whether $\overline{\sigma}$ satisfies φ, so we obtain non-conclusive verdict "?". In contrast, for $\neg\varphi = \neg p$, we have $tf(p)$ is not satisfied at t, then φ is not satisfied on the interval $[t-\delta; t+\delta]$ on $\rho_{\overline{\sigma}}$, then there is no possible timeline satisfying p at t. Hence, we can obtain guarantees that $\overline{\sigma}$ satisfies φ. As a conclusion, the fragments of the property that can be satisfied (resp. violated) at all time points in the interval $[t - \delta; t + \delta]$ and consequently by all (resp. none) of the timelines π and emit the verdict $\top$ (resp. $\bot$) are those in which atomic propositions occur only negatively.

4.2.4. Basin et al. [8]: RV of temporal properties over out-of-order data streams

Type of uncertainty targeted. Basin et al. present an approach for RV of properties over a data stream whose events may arrive to the monitor out of order or may not arrive due to delays and losses (this represents case 8 of Section 3.2.3).

Type of events and their representation. The monitor observes a prefix of a timed word with gaps due to arbitrary message delays. These gaps may be filled when more messages arrive to the monitor from time to time. The timed word is a sequence of letters, and each letter is of the form $\langle I, \sigma \rangle$ where I is a non-empty interval describing a time point in the timed word and σ is a partial function describing an action. Initially the monitor does not know anything about the system behavior, so the timed word is represented as an infinite gap $\langle [0, \infty), [] \rangle$. If a message (such as *"predicate p is true"*) arrives at timestamp 1, the interval $[0, \infty)$ will be split and the timed word becomes $\langle [0, 1), [] \rangle \langle \{1\}, [p \rightarrow true] \rangle \langle (1, \infty), [] \rangle$, and so on. If the monitor concludes that no action in the interval [0, 1), the letter $\langle [0, 1), [] \rangle$ can be removed and the timed word becomes $\langle \{1\}, [p \rightarrow true] \rangle \langle (1, \infty), [] \rangle$.

Method used to represent specification property and the verdict type. Basin et al. extend MTL (Section 2.3.4) into MTL$^{\downarrow}$ to reason about data values in the trace, where a freeze quantifier $\downarrow$ is used to take a value from a register in the state at a time point and freezes it into a variable. A freeze quantifier is a weak form of existential quantification. An MTL$^{\downarrow}$ policy example is:

$$\Box \downarrow^{cid} c. \downarrow^{tid} t. \downarrow^{amt} a.trans(c, t, a) \wedge a \geq 2000 \rightarrow$$
$$\Box_{(0,3]} \downarrow^{tid} t'. \downarrow^{amt} a'.\neg trans(c, t', a')$$

which states that "if a customer executes a transaction that exceeds $2000, then he must not execute any other transaction for 3 days". The registers *cid*, *tid* and *amt* stores the customer id, transaction id and the transferred sum respectively. The variables c and t are frozen to the values in *cid* and *tid* respectively. The variables a and a' are frozen to values stored in the register *amt* but at different times. The same for t and t'.

Basin et al. interpret the truth values as in Kleene logic and conservatively extend the logic's standard Boolean semantics as in [6]. *MTL*$^{\downarrow}$'s three-valued semantics is defined by $[[w, i, v \models \varphi]] \in \mathbb{B}_3$ where w is the observation, $i \in \mathbb{N}$ is the time point and $v : V \rightarrow D$ is a partial valuation that maps each logical variable to its value (V is the set of variables and D is the data domain). If $\varphi = t$ or $\varphi = f$, a precise verdict is simply produced. However, if $\varphi = p(\bar{x})$, then a precise verdict is produced only if $v(\bar{x})$ is defined, otherwise a non-conclusive verdict $\bot$ is emitted. If we have $\downarrow^r x.\varphi$, then the valuation v is obtained by freezing the value of x to the value in the register r. For the Boolean connectives $\neg$, $\vee$ and $\wedge$, the interpretation is trivial. However for other connectives such as $\mathbf{U}_I$, more interpretation is needed.

4.2.5. Ferrando et al. [9]: RV with imperfect information through indistinguishability relations

Type of uncertainty targeted. According to Ferrando et al., the standard RV of LTL properties is based upon the assumption that the absence of an event a is considered equivalent to its negation $\neg a$, which is not true in a case where a exists but it is indistinguishable from another event. So, they focus on differentiating between knowing when something is not true and knowing when something is unknown.

Type of events and their representation. Events are atomic propositions from an alphabet Σ. The absence of information is characterized by duplicating Σ such that $\bar{\Sigma} = \{p_\top, p_\bot, \forall p \in \Sigma\}$. The imperfect in information happens when atomic propositions such as p and q cannot be distinguished from each other (this represents case 2 od Section 3.2.3). This allows to introduce the equivalence relation $p \sim q$, the equivalent class $\gamma = \{p, q\}$, and the witness $[\gamma]_\top = \{p_\top, q_\top\}$ and $[\gamma]_\bot = \{p_\bot, q_\bot\}$.

They define two versions of traces: the *explicit* version σ_e where $p_\top \in \sigma_e(i)$ if p holds at $\sigma(i)$ and $p_\bot \in \sigma_e(i)$ if p does not hold at $\sigma(i)$; and the *visible* version σ_v derived from the σ_e where $[\gamma]_\top$ (resp. $[\gamma]_\top$) $\in \sigma_v(i)$ if $\forall p \in \gamma, p_\top$ (resp. $p_\bot$) $\in \sigma_e(i)$.

Method used to represent specification property and the verdict type. Ferrando et al. use LTL to express the property φ. They define an explicit version $\epsilon(\varphi)$ of φ, where $\epsilon(p) = [\gamma]_\top$ and $\epsilon(\neg p) = [\gamma]_\bot$. They also define the operators $\vee$ and $\wedge$, as well as the *next* operator $\circ$, where $\epsilon(\circ\varphi) = \circ\epsilon(\varphi)$.

Ferrando et al. extend the standard monitor's synthesis pipeline (Fig. 2 of Section 2.1.1) to explicitly consider imperfect information. They generate the DFA of $\epsilon(\varphi)$ to recognize the prefixes of trace that satisfy φ and $\epsilon(\neg\varphi)$ to recognize those that violate φ. However, the duplication of the atomic propositions in the formula breaks the duality between φ and $\neg\varphi$. For this reason, they added $\otimes\varphi$ which is $\neg\epsilon(\varphi) \wedge \neg\epsilon(\neg\varphi)$ and followed the same steps to generate the DFA of $\otimes\varphi$ which can recognize the prefixes having continuations that do not satisfy nor violate φ.

Each of the three monitors will process a visible trace δ_v and return a verdict in $\{\top, \bot, ?\}$. The resulting the three verdicts $v_{\epsilon(\varphi)}, v_{\epsilon(\neg\varphi)}, v_{\otimes\varphi}$ emitted by the DFA of $\epsilon(\varphi)$, DFA of $\epsilon(\neg\varphi)$ and DFA of $\otimes\varphi$ respectively, can be combined to deduce one final outcome. Five possible combinations exist: $\top$ if there is no continuation of δ_v which either violates φ or makes it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \bot$). The verdict $\bot$ if there is no continuation which either satisfies φ or makes it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \top, v_{\epsilon(\varphi)} = \bot$). The verdict *uu* if there is no continuation which either satisfies or violates $\epsilon(\varphi)$ (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \top$). The verdict $?_{\not\bot}$ if there is no continuation which will eventually violate $\epsilon(\varphi)$, but there are continuations that satisfy $\epsilon(\varphi)$ and make it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top, v_{\otimes\varphi} = \bot, v_{\epsilon(\varphi)} = \top$). Symmetrically, the verdict $?_{\not\top}$ if there are no continuations satisfying $\epsilon(\varphi)$, but continuations that violate $\epsilon(\varphi)$ and make it undefined exist (i.e. $v_{\epsilon(\neg\varphi)} = \bot, v_{\otimes\varphi} = \top, v_{\epsilon(\varphi)} = \top$).

Finally, the verdict ? if the monitor cannot conclude anything yet, because there exist continuations satisfying $\epsilon(\varphi)$, continuations violating $\epsilon(\varphi)$, and continuations that make it undefined (i.e. $v_{\epsilon(\neg\varphi)} = \top$, $v_{\otimes\varphi} = \top$, $v_{\epsilon(\varphi)} = \top$).

4.2.6. Aceto et al. [10]: Monitoring for silent actions

Type of uncertainty targeted. The approach conducted by Aceto et al. centers around the monitorability of a system that encounters *silent actions* or events. These actions refer to computational steps that are not revealed in the system model's level of abstraction. Nonetheless, the model presents sufficient indications of their occurrence throughout execution.

Type of events and their representation. Two types of actions are represented using atomic symbols: external or observable actions and silent actions. The system states or processes are modeled as a standard labeled-transition system (LTS) model L, where actions stimulate the transitions between states. The processing of silent action is represented by a τ-transition. Several silent actions can happen successively causing a sequence of τ-transitions which can be obscured by turning them into υ-transitions, thus hiding how many transitions were taking place at certain points and obtaining an obscured LTS L' (the obscuring of the number of transitions is equivalent to case 3 of Section 3.2.3). Any state having a τ-transition in L still have a τ-transition in L'. External transitions are not affected and if a state p has a sequence of τ-transitions in L leading to a state q that can perform an external action, this observation is preserved in L'.

Method used to represent specification property and the verdict type. Specification properties are expressed in a variant of the modal μ-calculus called μHML formulae (Hennessy Milner Logic) described in [137]. μ-HML is a dynamic logic with structure similar to an automaton and modal operators that also describe τ-transitions. Its operators include *true*, *false*, $\vee$, $\wedge$, $[\mu]\varphi$, p which states that $\forall$ state q of the LTS reached by event μ from p, φ holds at q, and $\langle\mu\rangle\varphi$, p which states that $\exists q$ reached by event μ from p and φ holds at q.

The monitoring setup is composed of an LTS system L and a monitor M consists of a set of states S_M and accepts external and silent actions. When the system produces a trace event μ that the monitor is able to analyze by transitioning from m to n, where $m, n \in S_M$, the constituent components of a monitored system $m \triangleleft p$ move in lockstep, where $m \triangleleft p$ means that the LTS is in state p when M is in state m. On the other hand, if M is unable to analyze an event μ, the monitored system still executes, but the monitor transitions to an *inconclusive state end*, where it remains for the rest of the computation.

Aceto et al. focus on rejection monitors to monitor *safety* fragments of the μHML formula, and use a state *no* to designate the rejection state. A monitor at state m rejects a process p in L if there exists a process q in L and a sequence of actions s such that the monitor ends in state *no* and the system is at state q after processing the trace s.

4.3. Statistical-based solutions

Another way to verify a property against a trace that contains uncertainty is to statistically compute the probability that the property is satisfied. In other words, the probability that a positive verdict is emitted.

4.3.1. Stoller et al.: RV with state estimation (RVSE)

Type of uncertainty targeted. Stoller et al. [54] account for missing events in a trace and present the RVSE algorithm which is based on a statistical model of the monitored system. The aim is to fill the gaps and predict the probability that a positive verdict is emitted when encountering a gap.

Type of events and their representation. Simple atomic symbols are used to represent the internal states of the system. A Hidden Markov Model (HMM) is used to represent the actual internal states of the system and can be learned from complete system traces using machine learning algorithms [138–141]. The presence of a gap is represented by the symbol $gap(L)$, where L is a probability distribution representing the length of the gap and $L(l)$ is the probability that the gap has length l (this represents case 3 of Section 3.2.3).

Using the forward algorithm, at each time point t, the system is in some internal (hidden) state s_i, it undergoes a change to a next state s_k according to a transition probability, and emits an observation symbol o_j according to an observation probability. The result is a sequence of observation symbols such as $O = O_1, O_2, \ldots, O_t$.

Method used to represent specification property and the verdict type. Stoller et al. use a DFA $\mathcal{M}$ to represent the property φ to be monitored. For each observation symbol emitted by the HMM, $\mathcal{M}$ moves from the current state to the next one. The sequence O satisfies the property if and only if it leaves the monitor $\mathcal{M}$ in an accepting state m_f when processing the last observation symbol O_t, and the probability of satisfaction is computed based on the transition and observation probabilities taking into account all ways of reaching the configuration in which the HMM is in state s_t and $\mathcal{M}$ is in state m_f. If a gap appears in the observation sequence when $\mathcal{M}$ is at state n and $\mathcal{H}$ is at state s_i, the observation symbol, say υ, emitted by s_i cannot be determined. In this case, the extended forward algorithm sums over all the possibilities that the monitor can move from a predecessor p to n by adding the probability of each observation symbol between p and n.

Another statistical-based approach is proposed by Zhou et al. [142]. They first learn an HMM and transform it to a Discrete Time Markov Chain (DTMC), which is a stochastic process in which the next state depends only on the current state, and not any historical states. However, instead of using the classical forward algorithm, they used Baum–Welch algorithm (reader can refer to [143] for more details about the algorithm) to model the system based on the previously observed target event sequence.

4.3.2. Kalajdzic et al. [11]: RV with particle filtering (RVPF)

Type of uncertainty targeted. The approach of Kaladjzic et al. is based on that of Stoller's et al. They also account for the presence of gaps in the trace. However, they introduce a technique for controlling the trade-off between runtime overhead and prediction accuracy.

Type of events and their representation. Similar to Stoller et al., the system states are represented using an HMM and each state emits an observation symbol. The symbol $gap(L)$ is used to denote a possible gap whose length is drawn from a probability distribution L over the natural numbers (this represents case 3 of Section 3.2.3). However, Kalajdzic et al. introduce a new type of events called *"peek events"*, which represent observations of parts of the program state, which are performed probabilistically at the end of a gap. Peek events help correct the movement errors introduced by using the HMM model during gaps. After each gap, a peek operation inspects a variable or a set of variables in the program state and returns an observation q_t. This information provided by a peek event helps to reduce the uncertainty in the monitor state after gaps, which in turn narrows down the monitor DFA's possible states.

Method used to monitor and fill gaps and the verdict type. Similar to Stoller et al., the goal is to calculate a probability that the system's behavior satisfies φ, i.e. to produce a probabilistic verdict. However, in contrast to Stoller et al., Kalajdzic et al. model

the composition of the HMM and DFA as a Dynamic Bayesian Network (DBN), which is a type of Bayesian network that relates the system state variables x_t and the monitor state variables s_t to each other and to the observation variables o_t as well as to their previous states x_{x-1} and s_{t-1} over adjacent time steps. If a gap is encountered at time t, a peek event q_t is produced at the end of the gap.

Kalajdzic et al. proposed the RVPF algorithm where the system state is represented by a set of particles. A particle is a hypothetical state of the system being modeled which represents a possible value or configuration of the system's state, and is often drawn from a probability distribution that reflects the uncertainty in the state estimation. The idea is to represent the system state with a large number of particles and use them to estimate the probability distribution of the true state of the system. Particle filtering (PF) is used with sequential importance resampling (SIR) to estimate the internal state of the DBN. The importance weight of each particle in a state is summed to estimate the probability of that state. When an observed event occurs, each particle selects a state transition to execute by sampling the joint transition probability distribution of the DBN. The particles are then redistributed among the states that provided the best prediction of the current observation. By utilizing the DBN structure and the current observation, SIR is used to decrease the variance of the PF and enhance its performance.

4.3.3. Wilcox et al. [12]: RV of stochastic, faulty systems

Type of uncertainty targeted. Wilcox et al.'s monitor a safety property over mixed stochastic systems (which consist of both hardware and software components) that may suffer from state uncertainty as they degrade due to hardware failure, and imprecise or unobserved future states due to the possible interactions of their components (a state could be missing or uncertain but treated as missing, so this refers to cases 1 and 2 of Section 3.2.3).

Method used to represent specification property. A safety constraint φ is written in LTL (see 2.3.3) which is converted into automata (see 2.3.2), mainly an NBA to automate the monitoring. However, NBA does not guarantee a complete transition function of the safety requirement. Hence, an NBA is converted into a deterministic BA.

Type of events and their representation and the monitoring method. The embedded system states are represented using Probabilistic Hierarchical Constraint Automata (PHCA) formalism. PHCA is similar to an HMM in the sense that it employs hidden states and probabilistic transitions. However, PHCA incorporates state constraints and a hierarchy of component automata. The system is modeled as a collection of individual PHCA components that communicate through shared variables. Each component is defined by discrete modes of operation, which represent both normal and faulty behavior. These modes can transition probabilistically or based on system commands, and can also be constrained by the modes of other components.

The states of the PHCA are: q_t: the safety state of the system at time t, defined as the state of the DBA that describes the safety constraint φ, x_t: system state at time t, c_t: system command at time t and z_t the observation at time t. If x_t is observable, q_t can be easily calculated from available information. Else, q_t cannot be known. However, one can estimate the probability that the system remains safe with φ by determining the probability distribution of the DBA state q_t, which is based on the history of observations ($z_{1:t}$) and commands ($c_{1:t}$). This probability distribution is called a belief state $B(y_t) = \sum_{x_t} P(q_t, x_t | z_{1:t}, c_{1:t})$, where $y_t = q_t \otimes x_t$.

The computation of the belief state over the BA is similar to the standard Forward algorithm for HMM belief state update. The subsequent state is predicted in a stochastic manner, taking into account the previous belief and transition probabilities of the models. This prediction is subsequently adjusted based on received observations. The observation probability ($P(z_t|x_t)$) and transition probability ($P(x_t|x_{t-1}, c_t)$) are both reliant on the physical system's model. In the case of an HMM, these probabilities are defined as a component of the system's model. However, for PHCA, these probabilities are determined by calculating the transition and observation probabilities of the specified components throughout the system.

5. Synthesis

In Section 4, various approaches that address the challenge of RV when uncertainty exists in the underlying trace are described. However, these approaches vary in terms of the types of uncertainty considered, the formalism used to represent events, the specification languages for property representation, the monitoring methods/algorithms employed, and the types of verdicts produced. In this section, we aim to analyze and compare these differences. Furthermore, we evaluate each approach based on key features that characterize RV, such as soundness, completeness, and monotonicity.

5.1. Events and uncertainty representation

Approaches in Section 4 account for different types of uncertainty in the trace: missing events whose content is unknown, imprecise events whose content is not completely defined, events with imprecise timestamps whose time of occurrence is not clear, or unordered events that arrive to the monitor in an unknown sequence. Table 1 specifies the different types of events uncertainty for each approach and how each approach represents uncertain events.

Taleb et al. (Section 4.1.1) replace a missing event with all possible valuations which means that each imprecise or missing event is replaced by a set of possible replacements. Wang et al. (Section 4.1.3) and Basin et al. (Section 4.2.4) both account for unordered events. Wang et al. replace the whole trace (which is an abstract trace) by the set of all possible sequences or all possible orderings of the events, whereas Basin et al. used the timed word $\langle [0, \infty), [\ ] \rangle$ representing an infinite gap. Whenever a new event arrives, the timed word is split to insert the event at a specific time point. Leucker et al. (Section 4.1.2) account for three types of uncertainty: a missing event at time point t represented as $\perp t$, imprecise event at t is represented as $\top$ and a gap (a segment of the abstract stream representing all combinations of events in terms of timestamps and values) is represented as $\smile$. Basin et al. (Section 4.2.3) also account for imprecise timestamps in a trace represented as a timed word. They assume a timestamp imprecision $\delta \geq 0$ and an imprecise timestamp is assumed to belong to $[\tau_i - \delta, \tau_i + \delta]$. Based on this, several timelines are obtained from one timed word.

Joshi (Section 4.2.1), Stoller (Section 4.3.1) and Kalajdzic (Section 4.3.2) only account for complete loss of events, and not uncertain events. They simply represent a missing event by a symbol. Joshi et al. use the symbol χ, whereas Stoller et al. and Kalajdzic et al. use the symbol $gap(L)$. Ferrando 4.2.5 accounts for imprecise or indistinguishable events. If an event p is indistinguishable, it is represented as p and $\neg p$. For Wilcox et al. (Section 4.3.3), a missing event is simply an unobserved state x in the system model. As to Aceto et al. (Section 4.2.6), their approach is limited to one kind of uncertainty which is replacing a group of τ silent actions with one less precise υ silent action.

Basin et al. (Section 4.2.2) use a model that represents uncertainties over event occurrences. A knowledge gap with regard to

Table 1
Different types of events uncertainty and their representation.

Approach	Missing events	Imprecise events	Imprecise timestamps	Unordered events	Representation
Taleb (Section 4.1.1)	✓	✓			All possible valuations
Leucker (Section 4.1.2)	✓	✓	✓		$\bot$t or $\top$t or $\smile$
Wang (Section 4.1.3)	✓			✓	All possible sequences
Joshi (Section 4.2.1)	✓				Symbol χ
Basin (Section 4.2.2)	✓	✓			?
Basin (Section 4.2.3)			✓		interval $[t-\delta; t+\delta]$
Basin (Section 4.2.4)				✓	⟨time interval , [ ]⟩
Ferrando (Section 4.2.5)		✓			p and $\neg p$ $\forall p \in \Sigma$
Aceto (Section 4.2.6)	✓				υ-transition
Stoller (Section 4.3.1)	✓				Symbol $gap(L)$
Kalajdzic (Section 4.3.2)	✓				Symbol $gap(L)$
Wilcox (Section 4.3.3)	✓				Unobservable state x

whether a predicate $r(a_1, \ldots, a_n)$ happened at τ is represented as $r(a_1, \ldots, a_n)$ =?. In the case of Basin (Section 4.2.3), they represent an event as a tuple where each timestamp is replaced by an interval of time.

5.2. Different forms of verdicts

RV is all about producing one precise verdict for each event. However, the presence of uncertain or missing events makes this challenging due to the inability of the monitor to precisely observe the event and correctly emit a verdict. Changing the format of representing events in a trace to account for the imprecision in their content may change the form (in terms of structure) and the type (in terms of number) of the output verdicts. Table 2 shows for each approach in Section 4, the type of the produced verdict and the form used to represent it.

Some approaches produce a set of verdicts instead of one verdict. Taleb et al. (Section 4.1.1) replace each input event by the set of possible valuations and emits a verdict for each valuation. The verdict produced is in the form of a set of verdicts rather than one single verdict. They also suggest quantifying each verdict by counting how many uni-traces projections of a multi-trace result in each verdict, producing a form of "probability" or "likelihood". Similarly for Leucker (Section 4.1.2), whose approach emits a verdict for each concrete event, resulting in a set of verdicts (abstract verdict) for an abstract event. The result is an abstract verdict representing a set of the verdicts of the concrete events.

Stoller (Section 4.3.1), Kalajdzic (Section 4.3.2) and Wilcox (Section 4.3.3) use a probabilistic model such as HMM and PHCA to represent the system states, and estimate their verdicts as a probability that an event satisfies the property. Other approaches rely on an extension of existing formalisms to define new verdicts that account for missing/uncertain events, and produce a single verdict, such as Joshi (Section 4.2.1) who extends the existing RV-LTL to produce the non-conclusive verdict ? when processing the symbol χ. Similarly, Ferrando (Section 4.2.5) extend the standard monitor's pipeline to include the verdicts $\{uu, ?_{\not\perp}, ?_{\not\top}, ?\}$ and explicitly consider imprecise events. Aceto (Section 4.2.6) focus on rejection monitors for safety fragments of their policy, which ends in state *no* if a process is rejected and in a non-conclusive state *end* if the event cannot be analyzed (meaning that the property is non-monitorable). Their approach is limited in that it only accounts for violations. However, it tackles issues related to monitorability which are not considered in the other approaches discussed in this survey.

The rest of the approaches in Table 2 produce a single verdict in $\mathbf{B}_3$ such as Basin et al. (Sections 4.2.2, 4.2.3, 4.2.4) and Wang et al. (Section 4.1.3). Wang et al. generate all the possible sequences (concrete traces) consistent with the given abstract trace and produce a concrete verdict for each concrete trace. However, the final verdict is a single verdict $\top$ if all the concrete verdicts are $\top$, $\bot$ if all concrete verdicts are $\bot$, or ? otherwise.

5.3. Soundness, completeness and monotonicity

Soundness, completeness, and monotonicity are important concepts used to evaluate the effectiveness of RV approaches. Soundness indicates how much confidence one can have in the output of monitor. It refers to the ability to produce a correct verdict indicating whether the monitored system has either satisfied or violated the specified property, without any ambiguity or uncertainty.

Completeness, on the other hand, refers to the ability of an RV approach to capture all relevant events or behaviors of a system during runtime without missing any violations or satisfactions of the expected properties [41]. In other words, a monitor is complete if it is able to return a conclusive verdict, whenever processing an event. The monitor should not return the non-conclusive symbol "?" unless it is unable to correctly analyze the event or if the event is missing or imprecise and the monitor is not able to access it. In both cases, the monitor is not able to detect a violation or a satisfaction. Hence, it is not complete. Note that the output symbol "?" indicates that the outcome generated by the monitoring process lacks the precision to yield a definitive judgment ($\top$ or $\bot$). Consequently, it fails to establish a decisive determination regarding whether a property is entirely satisfied or unequivocally violated at the end of the trace.

Consider the property *p is eventually true over the trace 'qqqq'*, the monitor does not return a conclusive verdict ($\top$ or $\bot$) because it is not able to observe any p event. However, the monitor may be able to return a conclusive verdict when observing subsequent events. Some approaches tried to come up with a more precise verdict to distinguish this case (the case where the monitor may be able to change its verdict relying on the future events) from the case where the monitor will not be able to change it. Joshi et al. [5] emits $\top_p$ or $\bot_p$ and Ferrando et al. [9] emits $?_{\not\perp}$ or $?_{\not\top}$. Their approaches also emit the non-conclusive verdicts "?" or *uu*. However, our perspective perceives all of these outcomes as imprecise determinations that hinder the monitor from achieving completeness.

Monotonicity indicates that the verdicts obtained do not retract as new events become available. In other words, if a RV approach makes a conclusion about the behavior of a system based on a set of observations, and then additional observations are made, the conclusion should not be invalidated.

In scenarios involving uncertain or missing events, guaranteeing the above concepts becomes challenging because the presence of knowledge gaps could limit the ability of the monitor to detect some violations or satisfactions of the specified properties, which affects the soundness and completeness of the monitor. On the other hand, one must ensure that the verdicts persist after closing knowledge gaps to guarantee the monotonicity. Table 3 states for each of the approaches of Section 4 whether it totally or partially guarantees soundness, completeness and monotonicity. The symbols in the table are explained as follows: ✓ indicates

Table 2
Different methods to represent verdicts.

Approach	One-verdict	Multi-verdicts	Probability	Form
Taleb (Section 4.1.1)		✓		Set of verdicts from {⊤, ⊥, ?}
Leucker (Section 4.1.2)		✓		Set of verdicts from {⊤, ⊥, ?}
Wang (Section 4.1.3)	✓			One value from {⊤, ⊥, ?}
Joshi (Section 4.2.1)	✓			One value from {⊤, $\top_p$, ?, $\bot_p$, ⊥}
Basin (Section 4.2.2)	✓			One value from {⊤, ⊥, ?}
Basin (Section 4.2.3)	✓			One value from {⊤, ⊥, ?}
Basin (Section 4.2.4)	✓			One value from {t, f, ⊥}
Ferrando (Section 4.2.5)	✓			One value from {⊤, ⊥, *uu*, $?_{\mathcal{L}}$, $?_{\mathcal{T}}$, ?}
Aceto (Section 4.2.6)	✓			One value from {*no*, *end*}
Stoller (Section 4.3.1)			✓	Probability of satisfaction
Kalajdzic (Section 4.3.2)			✓	Probability of satisfaction
Wilcox (Section 4.3.3)			✓	Probability of satisfaction

that the feature is totally covered, ✓ indicates that the feature is partially covered, ✗ indicates that the feature is not covered and $\mathcal{U}$ indicates that it is undetermined, i.e., impossible to precisely determine whether the feature is covered or not.

The statistical-based approaches (Stoller et al. (Section 4.3.1), Kalajdzic et al. (Section 4.3.2), Wilcox et al. (Section 4.3.3)), use a model of the system (an HMM or a PHCA) to estimate the probabilities of hidden states, fill the gaps and generate a sequence of observation symbols representing the most likely sequence of hidden states and emitted events. However, it is important to note that the accuracy of the imputed events depends on the accuracy of the HMM and the estimated probabilities. If the HMM does not accurately capture the underlying system behavior or the estimated probabilities are unreliable, the generated sequence may not accurately reflect the actual system behavior, leading to false positives (incorrectly reporting a violation) and false negatives (failing to report a violation). Therefore, guaranteeing soundness and completeness of the statistical-based approaches is challenging. The same can be with respect to the monotonicity, as the verdict's consistency cannot be guaranteed.

Taleb et al. (Section 4.1.1) assume that all valuations of the input multi-event are still valuations of the output multi-event (world-preserving proxy), hence the verdicts that are supposed to be emitted for the input valuations are preserved. Consequently, false verdicts are still emitted for violating events, and true verdicts are still emitted for valid events, which guarantees the soundness of the approach. With respect to monotonicity, as each missing event is replaced by the set of all possible replacements and the multi-verdict includes all the possible verdicts, this means the same multi-verdict will be produced for any replacement. Hence verdicts are preserved. Similarly, the approach of Leucker et al. (Section 4.1.2) states that the verdict of each concrete trace persists in the abstract trace. This feature guarantees the soundness and monotonicity of the approach. The approach of Wang et al. (Section 4.1.3) states that a true (resp. false) verdict is emitted when monitoring a property over an abstract trace if and only if all the concrete traces consistent with this abstract trace evaluate to true (resp. false). Otherwise, the outcome is uncertain. This guarantees the monotonicity of the approach because the same verdict will be produced for any possible replacement of events. The approach is also sound because it produces correct verdicts. However, the three approaches of Sections 4.1.1, 4.1.2 and 4.1.3 are not complete because, in some cases, an uncertain verdict is produced.

With respect to Joshi et al. (Section 4.2.1), the loss-tolerant monitor $\mathcal{M}$ guarantees soundness because it produces a verdict at the end of the trace compatible with that of an RV-LTL monitor, assuming that a loss tolerant cluster and a loss tolerant alphabet exist. However, some patterns such as $\Box(a \rightarrow (b \wedge c))$ cannot be soundly monitored under transient loss. The approach guarantees monotonicity because, as described by Joshi et al., the output of the $\mathcal{M}$ is always equal to that of an RV-LTL (before and after processing the lossy elements χ). Since the outputs of an RV-LTL monitor is monotonic, we conclude that the output of $\mathcal{M}$ is also monotonic. However, the approach is not complete since the state machine produces the uncertain verdict ? when processing χ.

The approach of Basin et al. (Section 4.2.2) aims to avoid reporting a policy violation unless there is indeed a violation. This implies that the approach is sound. Their approach is not complete in the sense that some policy violations may not be reported. However, completeness is guaranteed on an expressive fragment of the compliance policy that retains all the language's connectives but limits the usage of free variables. With respect to the monotonicity requirement, the policy language used by Basin et al.'s work ensures that this requirement is maintained. In this language, evaluations of formulas do not reduce the amount of knowledge when resolving incompleteness in the extension of a logging knowledge base.

The approach of Basin et al. (Section 4.2.4) provides soundness and completeness guarantees in the sense that verdicts are correct w.r.t. the observations given to the monitor, meaning that, assuming no failures occur, violations and satisfactions of specifications will eventually be reported despite the presence of finite message delays. Their reasoning is monotonic with respect to the partial order on truth values, where ⊥ is less than t and f, and t and f are incomparable. This monotonicity property ensures that closing knowledge gaps does not contradict previously obtained Boolean truth values. In other words, when filling a knowledge gap represented by ⊥ with either t or f, the resulting truth value will always be consistent with the previously obtained one.

The approach of Basin et al. (Section 4.2.3) is sound in the sense it always emits a correct verdict. However, soundness is guaranteed only for certain MTL fragments in which atomic propositions occur only negatively. The approach is also complete for these fragments because the same precise verdict is emitted for all the timelines and for the timed word $\overline{\sigma}$. Similar to Basin et al. (Section 4.2.4), the approach is monotonic.

The approach of Ferrando et al. (Section 4.2.5) is sound in the sense that all the emitted verdicts are correct. In other words, a negative verdict is emitted only if a violation occurs and a positive verdict is emitted only if a satisfaction happens. However, the algorithm is not complete in the sense that at some point, no verdict is emitted (represented by "?″) which means that a satisfaction or a violation is missed. Monotonicity is guaranteed because the verdict ⊤ is produced if and only if there is no continuation of δ_v which either violates φ or makes it undefined, and the verdict ⊥ is produced if and only if there is no continuation which either satisfies φ or makes it undefined. This means that once the verdict ⊤ or ⊥ is emitted, it persists over all the possible continuations of the trace.

According to Aceto et al. (Section 4.2.6), their monitor can check for a μHML formula φ on L from any obscuring L' of L if $\forall p$ in L': p does not satisfy φ on L if and only if p is rejected by the monitor on L'. So, the verdict produced when monitoring over L'

Table 3
Features and limitations in related works.

Approach	Monotonic	Complete	Sound
Taleb (Section 4.1.1)	✓	✗	✓
Leucker (Section 4.1.2)	✓	✗	✓
Wang (Section 4.1.3)	✓	✗	✓
Joshi (Section 4.2.1)	✓	✗	✓
Basin (Section 4.2.2)	✓	✓✗	✓
Basin (Section 4.2.3)	✓	✓✗	✓
Basin (Section 4.2.4)	✓	✓✗	✓
Ferrando (Section 4.2.5)	✓	✗	✓
Aceto (Section 4.2.6)	✓	✓✗	✓
Stoller (Section 4.3.1)	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$
Kalajdzic (Section 4.3.2)	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$
Wilcox (Section 4.3.3)	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$

Table 4
Different methods to represent events and policies.

Approach	Event type	Policy
Taleb (Section 4.1.1)	Valuation over Boolean variables	DFA
Leucker (Section 4.1.2)	Timestamp, data value	TeSSLa
Wang (Section 4.1.3)	Atomic event or process variable	Past-LTL
Joshi (Section 4.2.1)	Atomic symbol	LTL
Basin (Section 4.2.2)	Predicate with arity	FOTL
Basin (Section 4.2.3)	Tuple (atomic symbol, timestamp)	MTL
Basin (Section 4.2.4)	Tuple (time interval, atomic symbol)	Freeze MTL
Ferrando (Section 4.2.5)	Atomic symbol	LTL
Aceto (Section 4.2.6)	Atomic symbol	μHML Logic
Stoller (Section 4.3.1)	Observation symbol	DFA
Kalajdzic (Section 4.3.2)	Observation symbol	DFA
Wilcox (Section 4.3.3)	Observation symbol	LTL

is compatible with the verdict produced when monitoring over L. Hence, the approach is guaranteed to be sound. However, similar to Basin et al. (Section 4.2.2), completeness is guaranteed only for a fragment of the μHML formula. Aceto et al. state that once the monitor transitions to the inconclusive state *end* (resp. rejection state *no*), it remains in this state for the rest of the computation. This indicates that the approach is monotonic.

5.4. Comparison based on specification language

The approaches in Section 4 use various specification languages to express the specification property. The languages are summarized in Table 4. Each specification language is characterized by its operators and expressiveness. Some approaches such as that of Joshi et al. (Section 4.2.1), Ferrando et al. (Section 4.2.5) and Wilcox et al. (Section 4.3.3) simply use the LTL formalism to express properties using atomic propositions and Boolean connectives. To automate the monitoring process Joshi et al., Ferrando et al. and Wilcox et al. convert the LTL into FSM, DFA and BA respectively to automate the monitoring process.

Others such as Stoller et al. (Section 4.3.1), Kalajdzic et al. (Section 4.3.2) and Taleb et al. (Section 4.1.1) directly use finite state machines to represent the property. Aceto et al. (Section 4.2.6) use μHML formulae (Hennessy Milner Logic) whose structure is similar to an automaton. Additionally, it has modal operators that describe τ-transitions.

Some approaches use an extension of LTL such as Wang et al. (Section 4.1.3) who use Past-LTL which augments the LTL with operators that reason about the past. While Past-LTL does not offer greater expressiveness than LTL, it is much more concise and convenient for defining correctness properties when it comes to RV over finite traces [1]. Since the events in their approach lack explicit timestamps, only linear time properties in LTL are analyzed. Basin et al. (Section 4.2.2) also propose an augmented LTL specification language that use the operators of LTL and propose more connective operators to reason about the past and future time points and operators to reason about incompleteness and handle inconsistencies. Another language used by Basin et al. (Section 4.2.3) is MTL which extends LTL with timing constraints over the temporal operators to reason about the imprecision in timestamps. Later, Basin et al. (Section 4.2.4) extended the MTL with freeze variables to reason about data values in the trace. MTL and Freeze MTL have more operators than LTL and allow specifying time constraints. However, the presence of freeze quantifiers and temporal connectives in the specification property increases the running time of the monitoring algorithm.

The above logics are common in static verification and are not suitable for stream RV. In contrast, Leucker et al. (Section 4.1.2) extended the existing TeSSLa specification language into Abstract TeSSLa and propose an abstract operator for each concrete operator of TeSSLa. Their language is suitable for monitoring streams and is equipped with operators to reason about imprecise timestamps which increases its expressiveness.

The relative expressiveness of these languages cannot be established in a clear-cut manner. It is known that propositional LTL, past LTL and DFA are equivalent for finite prefixes of a trace. The remaining specification languages are strictly more expressive than those three, although a strict ordering between them is not known.

5.5. Comparison based on evaluation methods

A last element of comparison between these works is the empirical assessment of their performance. Table 5 summarizes the methods that each of the approaches in Section 4 rely on to evaluate their work and the results they obtained.

Taleb et al. (Section 4.1.1) run tests across a variety of uncertainty scenarios to determine the overhead imposed by the existence of the access proxy and multi-monitor in terms of both running time and memory consumption. In terms of running time, the presence of an access proxy causes a slowdown in the monitoring process because the monitor must handle multi-events rather than uni-events and track the many possible states of the uni-monitor. However, for the given scenarios, this slowing spans between $2\times$ and $8\times$, indicating that handling multi-events does not impose a significant burden on monitor performance. In terms of memory consumption, having many events increased the maximum amount of memory consumed by the monitor, but this increase is minor and never reaches a factor of 1.5.

Leucker et al. (Section 4.1.2) perform empirical evaluation on different TeSSLa specifications to evaluate the computational overhead in terms of how many concrete TeSSLa operators are needed to realize the Abstract TeSSLa specification. Results showed that evaluating the abstract specification typically only increases the computational cost by a constant factor, and if a concrete specification can be monitored in linear time (in the size of the trace) its abstract counterpart can be as well.

Wang et al. (Section 4.1.3) test a number of properties over an actual number of traces. The experiments show that 97.7% of the running time was spent on executing the *checkOne* function, due to the exponential number of concrete traces corresponding to an abstract state. With respect to the frequency of the resulting uncertain verdicts, the results show that a low number of traces (15.61%) end in inconclusive verdict. This is justified by the fact that most of the temporal operators are insensitive to the uncertainty, and also the scope of uncertainty is bounded within one abstract state.

Joshi et al. (Section 4.2.1) show that the additional overhead incurred by loss-tolerant monitor $\mathcal{M}$ due to additional states is

Table 5
Empirical evaluations of different approaches.

Approach	Evaluation factor	Result
Taleb (Section 4.1.1)	Running time	Between 2× and 8×
	Memory consumption	Less than 1.5
Leucker (Section 4.1.2)	Computational cost	Increased by constant factor
Wang (Section 4.1.3)	Running time of an abstract state	97% of the total running time
	Frequency of uncertain verdicts	15.61%
Joshi (Section 4.2.1)	Memory consumption	Between 5 and 534 transitions
	Time complexity	$m \times n \times 2^n$
	Complexity	$O(N_h^2 \times N_d)$
Basin (Section 4.2.4)	Running time	Increases rapidly
Ferrando (Section 4.2.5)	Monitor execution time	Linear w.r.t. the trace length
	Monitor synthesis time	Exponential w.r.t. LTL length
Stoller (Section 4.3.1)	Inaccuracy	15× better than naive approach
	Time complexity without gap	$O(N_h^2 \times N_d)$
	Time complexity with gap	$O(N_h^2 \times N_d^2)$
Kalajdzic (Section 4.3.2)	Memory consumption	$16 \times N_p + 3560$
	Execution time	Outperforms RVSE of Stoller
Wilcox (Section 4.3.3)	Time complexity	$O(n^2)$
	Space complexity	$O(n)$

not significant (only an increase of at most two from that of the corresponding RV-LTL monitor). The overhead in terms of memory at runtime due to the increased number of transitions in $\mathcal{M}$ is also proved to be minimal (fluctuating between 5 and 534 extra transitions) compared to an RV-LTL. With respect to the time complexity of the monitoring algorithm, it is exponential with the number of states of $\mathcal{M}$ and is equivalent to $m \times n \times 2^n$ where m is the size of Σ and n is the number of state in $\mathcal{M}$. It is also proved that the size complexity of $\mathcal{M}$ is identical to that of the RV-LTL monitor.

Basin et al. (Section 4.2.4) perform experiments to evaluate the effect of freeze quantifiers and temporal connectives in the specification property on the running time. They also offset the arrival time of an event by a random delay to evaluate the effect of out-of-order arrival on the running time. The results show an increase in the running time for formulas containing more freeze quantifiers and temporal connectives. The running time also increases when messages are received out-of-order.

Ferrando et al. (Section 4.2.5) carried out experiments by varying the length of the trace of events. The results showed that the execution time is linear w.r.t. the length of the trace, then the time required for the monitor to analyze a single event in the trace is constant. They also measure the execution time for the monitor synthesis from LTL formulas with different lengths and proved that increases exponentially with the size of the input formula.

Stoller et al. (Section 4.3.1) measure the overall inaccuracy (i.e. how many events are not observed due to sampling), and obtained a ratio of 0.0205. This level of inaccuracy is quite low, considering the severity of the sampling. In comparison, the inaccuracy of a naive approach that ignores gaps due to sampling is 0.3135; this is approximately 15× worse. With respect to time complexity, it is $O(N_h^2 \times N_d)$ for a single observation without a gap event and $O(N_h^2 \times N_d^2)$ for a gap event, where N_h and N_d are the numbers of states of the HMM and the DFA, respectively.

Kalajdzic et al. (Section 4.3.2) conduct experiments to evaluate the effect of the number of particles N_p on execution time and memory consumption. The amount of required memory is a linear function of the number of particles and was measured to be $16 \times N_p + 3560$ using a 10-state HMM. Compared to RVSE, this is higher, and in comparing to the AP-RVSE, it is around 80 times lower. In terms of execution time, RVPF outperforms RVSE for all gap lengths with increasing number of particles.

Wilcox et al. (Section 4.3.3) The cost of computing that a state is safe is entirely dependent on the sizes of Q and X. In order to find the probability of each q_t, we must loop twice over these sets. If n is the size of the combined set, $n = |Q \times X|$, then we have a time complexity of $O(n^2)$, and a space complexity of $O(n)$.

With respect to the rest of the approaches in Table 5 (Basin et al. Sections 4.2.2 and 4.2.3 and Aceto et al. Section 4.2.6), no empirical evaluations are provided.

6. Conclusion and future work

This survey provided a comprehensive analysis of the existing literature on monitoring with incomplete traces. It discussed the various sources of uncertainty that have been identified and examines their impact on the monitoring process. The survey evaluated and compared different approaches for monitoring incomplete traces, taking into consideration the types of uncertainties addressed, representations of uncertain events, the formalism used for event and policy representation, and the methods of representing output verdicts. The advantages and limitations of each approach were also highlighted based on their respective evaluation results. The thorough analysis of the surveyed works allows us to identify several areas where future research is warranted. We list the main ones in the following.

First, a direct extension of the model proposed by Taleb et al. (Section 4.1.1) would be the symbolic manipulation of infinite or continuous variables, which would allow the convenient expression of a wider range of event types and access restrictions. Additionally, the concept of uncertainty could be broadened to encompass other formal notations beyond Mealy machines, such as Linear Temporal Logic. Another potential avenue for future research lies in the realm of runtime enforcement [144,145], where the proxy can be modeled to make the modifications so that each trace it produces is a replacement of the input trace that satisfies the given property, and all output traces could then be evaluated to identify the optimal replacement trace that requires the minimum number of modifications.

The approach developed by Wang et al. (Section 4.1.3) can be expanded to incorporate the complete semantics of LTL, including past-time and future-time operators. However, this extension would require significant effort. Another potential extension could address imprecise timestamps in recorded traces. Indeed, in the recorded traces, abstract states are timestamped when the state is recorded, but the time of actual observations is lost, which introduce uncertainty for timed operators. Therefore, it would be valuable to explore techniques for handling imprecise timestamps and addressing the impact of uncertainty on real-time properties.

Joshi et al. (Section 4.2.1) research could be extended to allow the approach to handle timed traces and more complex system architectures, such as distributed and concurrent systems. The approaches of Basin et al. ((Section 4.2.2) and (Section 4.2.3)) could be enhanced by conducting case studies to evaluate their

effectiveness in real-world settings. Similarly, Aceto et al. (Section 4.2.6,) could assess the performance of their proposed monitoring approach by proposing a monitoring algorithm and conducting experiments. Basin et al. (Section 4.2.2) would also explore different truth spaces to distinguish between different kinds of knowledge gaps and disagreements.

While the experimental evaluation of Basin et al.'s approach (Section 4.2.4) is promising, their method currently cannot handle the monitoring of systems generating thousands or millions of events per second. Further research is necessary, including algorithmic optimization, which the authors plan to undertake in the future, as well as deploying and evaluating their approach in large-scale case studies. Finally, future investigations could focus on the empirical assessment of the time and space complexity of the monitoring process. Ferrando et al. (Section 4.2.5) plan to expand their approach in future work by proposing a method to add additional information to the monitor's verdict. This method would utilize the event trace, the LTL property, and the monitor's verdict to establish a level of confidence in the final outcome. Specifically, in cases where the outcome is *uu*, instead of simply stating that the property is undefined with respect to the trace, they could use a probability distribution over the relevant equivalence classes to assert that the property would be satisfied (or violated) with a certain probability threshold.

In the approach of Stoller et al. (Section 4.3.1), the matrix–vector calculations performed by the RVSE algorithm to get the transition and observation probabilities when processing any observation symbol makes the computation cost very high and increase the overhead dramatically, especially in the presence of large gaps. One future direction is to tackle this problem. Bartocci et al. [146] propose *approximately precomputed* RVSE (AP-RVSE) that significantly reduces the runtime overhead of RVSE by precomputing and storing the results of the matrix calculations performed by RVSE. However, their approach introduces some approximation error. With respect to Kalajdzic et al. (Section 4.3.2), an interesting extension could involve creating a runtime-variable version of RVPF, in which the number of particles employed for state estimation can be adjusted dynamically. This would enable the flexible control of the tradeoff between estimation accuracy, memory consumption, and speed. The approach of Wilcox et al. (Section 4.3.3) has undergone preliminary validation, which demonstrates its ability to rapidly and precisely identify safety violations in small models. Their future efforts might focus on determining the effectiveness of these methods on larger models.

Taleb et al. (Section 4.1.1) is the only work with an explicit modeling of noise, in the form of the proxy, which makes it possible to model various kinds of perturbations (or data degradation) on an input trace and observe their effect on the monitor. For example, a valuation can simply swap the assignments of events a and b to make them indistinguishable: an input multi-event that supports a is transformed into an output multi-event that only supports the weaker proposition $a \vee b$ (and similarly for events that support b). In other words, it is no longer possible to conclude precisely that a is true or that b is true, only that at least one of them is true. Ferrando et al. (Section 4.2.5) can represent this form of uncertainty as an equivalence relation $a \sim b$. Some language-based approaches do not account for uncertain events (e.g. Sections 4.2.3, 4.2.4) and other approaches (Sections 4.2.1, 4.2.2, 4.2.6) are limited in their ability to account for uncertainty. The best these approaches can do is to approximate uncertainty by assuming that the occurrence of both a and b is unknown. However, this abstraction is only precise for events where neither a nor b are true. When dealing with abstract data domains such as that of Leucker et al. (Section 4.1.2), the situation becomes even less desirable. These domains are defined for each variable separately and must remain consistent throughout the entire trace. Therefore, the only way to preserve the world when abstracting is to replace the values of a and b with all possible values at all time points, resulting in an even greater over-approximation. Statistical-based approaches (Sections 4.3.1, 4.3.2 and 4.3.3) can be also extended to deal with imprecise events. An imprecise event $a \vee b$ can be treated in the same way as a missing event (gap), however, assuming the monitor is at state n, one can add only the transition probabilities of the predecessors of n where $a \vee b$ holds instead of summing over all the predecessors of n.

Each of the approaches surveyed should consider addressing the problem of incompleteness as an additional future extension. Table 3 shows that all of the approaches surveyed (except the statistical-based ones) guarantee soundness. However, most of them are not complete. Even those considered as complete (Sections 4.2.2, 4.2.3, 4.2.4, 4.2.6) provide completeness only for some fragments of their specification policy.

An interesting future extension of the RV approaches under uncertainty is to study the effect of uncertainty on the monitored property. A property could be robust with respect to the existing type of uncertainty, i.e. still produces the correct verdict despite the imperfect events in the trace. For example, the property "every a is eventually followed by c" is robust to a type of trace corruption where events b and d are indistinguishable. One could also modify the property to make it more robust. In the work of Alechina et al. [147] for example, instead of modifying the trace to enhance the observation capabilities, they show how to synthesize an approximation of an "ideal" norm that can be perfectly monitored given a monitor, and which is optimal in the sense that any other approximation would fail to detect at least as many violations of the ideal norm.

An RV approach could be also improved by creating a specification formalism that provides explicit constructs to express constraints on the system's behavior that take into account the possibility of imprecise events directly from within the property. For example, a property can constrain imprecise events to correspond to at most n concrete events which help ensure that the system is able to maintain a desired level of accuracy or precision in its behavior. This would also minimize the number of possible replacement traces of the input trace and the number of output verdicts. The specification formalism could also specify that no trace should contain more than n successive missing events. By imposing this limit, the specification can ensure that the system is able to recover from errors or unexpected inputs. By including explicit constructs for reasoning about uncertainty and imprecision, the specification formalism may be able to provide more precise and flexible ways to specify requirements for RV of complex systems. However, it is important to carefully design and validate such constructs to ensure that they are useful and effective in practice.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Sylvain Halle reports financial support was provided by Canada Research Chairs. Sylvain Halle reports financial support was provided by Natural Sciences and Engineering Research Council of Canada. Raphael Khoury reports financial support was provided by Natural Sciences and Engineering Research Council of Canada.

Funding

Canada Research Chairs, grant number 950-230760.

Data availability

No data was used for the research described in the article.

References

[1] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society, 1977, pp. 46–57, http://dx.doi.org/10.1109/SFCS.1977.32.

[2] R. Taleb, R. Khoury, S. Hallé, Runtime verification under access restrictions, in: 9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021, IEEE, 2021, pp. 31–41, http://dx.doi.org/10.1109/FormaliSE52586.2021.00010.

[3] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Methods Program 78 (5) (2009) 293–303, http://dx.doi.org/10.1016/j.jlap.2008.08.004.

[4] S. Wang, A. Ayoub, O. Sokolsky, I. Lee, Runtime verification of traces under recording uncertainty, in: Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011. Revised Selected Papers, 2011, pp. 442–456, http://dx.doi.org/10.1007/978-3-642-29860-8_35.

[5] Y. Joshi, G.M. Tchamgoue, S. Fischmeister, Runtime verification of LTL on lossy traces, in: A. Seffah, B. Penzenstadler, C. Alves, X. Peng (Eds.), Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017, ACM, 2017, pp. 1379–1386, http://dx.doi.org/10.1145/3019612.3019827.

[6] D.A. Basin, F. Klaedtke, S. Marinovic, E. Zalinescu, 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, in: Vol. 7687 of Lecture Notes in Computer Science, Springer, 2012, pp. 151–167, http://dx.doi.org/10.1007/978-3-642-35632-2_17.

[7] D.A. Basin, F. Klaedtke, S. Marinovic, E. Zalinescu, 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, in: Vol. 8734 of Lecture Notes in Computer Science, Springer, 2014, pp. 193–198, http://dx.doi.org/10.1007/978-3-319-11164-3_16.

[8] D.A. Basin, F. Klaedtke, E. Zalinescu, Runtime verification of temporal properties over out-of-order data streams, in: R. Majumdar, V. Kuncak (Eds.), Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017. Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 10426, Springer, 2017, pp. 356–376, http://dx.doi.org/10.1007/978-3-319-63387-9_18.

[9] A. Ferrando, V. Malvone, Runtime verification with imperfect information through indistinguishability relations, in: B. Schlingloff, M. Chai (Eds.), Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022. Proceedings, in: Lecture Notes in Computer Science, vol. 13550, Springer, 2022, pp. 335–351, http://dx.doi.org/10.1007/978-3-031-17108-6_21.

[10] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, Monitoring for silent actions, in: 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017. Kanpur, India, 2017, pp. 7:1–7:14, http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2017.7.

[11] K. Kalajdzic, E. Bartocci, S.A. Smolka, S.D. Stoller, R. Grosu, Runtime verification with particle filtering, in: Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013, Proceedings, 2013, pp. 149–166, http://dx.doi.org/10.1007/978-3-642-40787-1_9.

[12] C.M. Wilcox, B.C. Williams, Runtime verification of stochastic, faulty systems, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), in: Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 452–459, http://dx.doi.org/10.1007/978-3-642-16612-9_34.

[13] K. Havelund, G. Reger, D. Thoma, E. Zalinescu, Monitoring events that carry data, in: E. Bartocci, Y. Falcone (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics, in: Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018, pp. 61–102, http://dx.doi.org/10.1007/978-3-319-75632-5_3.

[14] M. Pezze, M. Young, A survey of software testing techniques, ACM Comput. Surv. 40 (4) (2008) 1–92.

[15] E.M. Clarke, O. Grumberg, D.E. Long, Model checking, in: M. Broy (Ed.), Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany, 1996, pp. 305–349.

[16] Y. Bertot, P. Castéran, Interactive theorem proving and program development - Coq'art: The calculus of inductive constructions, in: Texts in Theoretical Computer Science, in: An EATCS Series, Springer, 2004, http://dx.doi.org/10.1007/978-3-662-07964-5.

[17] W. Xu, L. Huang, A. Fox, D.A. Patterson, M.I. Jordan, Detecting large-scale system problems by mining console logs, in: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009, 2009, pp. 117–132, http://dx.doi.org/10.1145/1629575.1629587.

[18] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, S. Pasupathy, SherLog: Error diagnosis by connecting clues from run-time logs, in: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010, 2010, pp. 143–154, http://dx.doi.org/10.1145/1736020.1736038.

[19] D.P. Attard, A. Francalanza, 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, in: Vol. 10012 of Lecture Notes in Computer Science, Springer, 2016, pp. 473–481, http://dx.doi.org/10.1007/978-3-319-46982-9_31.

[20] K. Havelund, G. Reger, Runtime verification logics a language design perspective, in: L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, R. Mardare (Eds.), Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, in: Vol. 10460 of Lecture Notes in Computer Science, Springer, 2017, pp. 310–338, http://dx.doi.org/10.1007/978-3-319-63121-9_16.

[21] M. Laurenzano, M.M. Tikir, L. Carrington, A. Snavely, PEBIL: Efficient static binary instrumentation for linux, in: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA, IEEE Computer Society, 2010, pp. 175–183, http://dx.doi.org/10.1109/ISPASS.2010.5452024.

[22] E. Bodden, K. Havelund, Racer: Effective race detection using aspectj, in: B.G. Ryder, A. Zeller (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, ACM, 2008, pp. 155–166, http://dx.doi.org/10.1145/1390630.1390650.

[23] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (iot): A vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660, http://dx.doi.org/10.1016/j.future.2013.01.010.

[24] G. Reger, K. Havelund, What is a trace? A runtime verification perspective, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016. Proceedings, Part II, in: Vol. 9953 of Lecture Notes in Computer Science, 2016, pp. 339–355, http://dx.doi.org/10.1007/978-3-319-47169-3_25.

[25] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64, http://dx.doi.org/10.1145/2000799.2000800.

[26] K. Havelund, G. Rosu, Efficient monitoring of safety properties, Int. J. Softw. Tools Technol. Transf. 6 (2) (2004) 158–173, http://dx.doi.org/10.1007/s10009-003-0117-6.

[27] J. Huang, C. Erdogan, Y. Zhang, B.M. Moore, Q. Luo, A. Sundaresan, G. Rosu, ROSRV: runtime verification for robots, in: B. Bonakdarpour, S.A. Smolka (Eds.), Runtime Verification - 5th International Conference, RV 2014, Toronto, on, Canada, September 22-25, 2014 Proceedings, in: Vol. 8734 of Lecture Notes in Computer Science, Springer, 2014, pp. 247–254, http://dx.doi.org/10.1007/978-3-319-11164-3_20.

[28] A. Artikis, T. Eiter, A. Margara, S. Vansummeren, Foundations of composite event recognition Dagstuhl seminar 20071, Dagstuhl Rep. 10 (2) (2020) 19–49, http://dx.doi.org/10.4230/DagRep.10.2.19, https://drops.dagstuhl.de/opus/volltexte/2020/13058.

[29] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstic, J.M. Lourenço, D. Nickovic, G.J. Pace, J. Rufino, J. Signoles, D. Traytel, A. Weiss, A survey of challenges for runtime verification from advanced application domains (beyond software), Formal Methods Syst. Des. 54 (3) (2019) 279–335, http://dx.doi.org/10.1007/s10703-019-00337-w.

[30] A. Mrad, S. Ahmed, S. Hallé, É. Beaudet, Babeltrace: A collection of transducers for trace validation, in: S. Qadeer, S. Tasiran (Eds.), Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, in: Vol. 7687 of Lecture Notes in Computer Science, Springer, 2012, pp. 126–130, http://dx.doi.org/10.1007/978-3-642-35632-2_14.

[31] R. Gerth, D.A. Peled, M.Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: P. Dembinski, M. Sredniawa (Eds.), Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification,

Testing and Verification, Warsaw, Poland, 1995, 38 of IFIP Conference Proceedings, Chapman & Hall, 1995, pp. 3–18.
[32] M.O. Rabin, D.S. Scott, Finite automata and their decision problems, IBM J. Res. Dev. 3 (2) (1959) 114–125, http://dx.doi.org/10.1147/rd.32.0114.
[33] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, J. Comput. System Sci. 78 (3) (2012) 911–938, http://dx.doi.org/10.1016/j.jcss.2011.08.007.
[34] M.Y. Vardi, P. Wolper, Automata-theoretic techniques for modal logics of programs, J. Comput. Syst. Sci. 32 (2) (1986) 183–221.
[35] M.Y. Vardi, Automatic verification of probabilistic concurrent finite-state systems, Distrib. Comput. 11 (3) (1998) 139–155.
[36] T. Babiak, F. Blahoudek, J. Křet'ınsk'y, D. Štill, Ltl2dstar: A tool for ltl synthesis, in: Proceedings of the 24th International Conference on Computer Aided Verification, Springer, 2012, pp. 571–577.
[37] B. Finkbeiner, M. Schewe, Efficient translation of ltl formulae into deterministic Büchi automata, in: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, 2006, pp. 53–67.
[38] D. D'Souza, P. Thiagarajan, Synthesis of non-deterministic automata from temporal logic specifications, Form. Methods Syst. Des. 17 (1) (2000) 5–30.
[39] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: E. Bartocci, Y. Falcone (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics, in: Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018, pp. 1–33, http://dx.doi.org/10.1007/978-3-319-75632-5_1.
[40] C. Luk, R.S. Cohn, R. Muth, H. Patil, A. Klauser, P.G. Lowney, S. Wallace, V.J. Reddi, K.M. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: V. Sarkar, M.W. Hall (Eds.), Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, ACM, 2005, pp. 190–200, http://dx.doi.org/10.1145/1065010.1065034.
[41] Y. Falcone, S. Krstic, G. Reger, D. Traytel, A taxonomy for classifying runtime verification tools, in: C. Colombo, M. Leucker (Eds.), Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, in: Vol. 11237 of Lecture Notes in Computer Science, Springer, 2018, pp. 241–262, http://dx.doi.org/10.1007/978-3-030-03769-7_14.
[42] A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly? in: O. Sokolsky, S. Tasiran (Eds.), Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers, in: Vol. 4839 of Lecture Notes in Computer Science, Springer, 2007, pp. 126–138, http://dx.doi.org/10.1007/978-3-540-77395-5_11.
[43] F.B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50, http://dx.doi.org/10.1145/353323.353382.
[44] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Inform. Syst. Secur. 12 (3) (2023) http://dx.doi.org/10.1145/1455526.1455532.
[45] Y. Falcone, You should better enforce than verify, in: Runtime Verification - First International Conference, RV 2010 St Julians, Malta, November 1-4, 2010, Proceedings, in: 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 89–105, http://dx.doi.org/10.1007/978-3-642-16612-9_9.
[46] M. d'Amorim, K. Havelund, Event-based runtime verification of Java programs, ACM SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–7, http://dx.doi.org/10.1145/1082983.1083249.
[47] M.R. Boussaha, R. Khoury, S. Hallé, Monitoring of security properties using beepbeep, in: A. Imine, J.M. Fernandez, J. Marion, L. Logrippo, J. García-Alfaro (Eds.), Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers, in: Vol. 10723 of Lecture Notes in Computer Science, Springer, 2017, pp. 160–169, http://dx.doi.org/10.1007/978-3-319-75650-9_11.
[48] J. Simmonds, S. Ben-David, M. Chechik, Monitoring and recovery of web service applications, in: M.H. Chignell, J.R. Cordy, J. Ng, Y. Yesha (Eds.), The Smart Internet - Current Research and Future Applications, in: Vol. 6400 of Lecture Notes in Computer Science, Springer, 2010, pp. 250–288, http://dx.doi.org/10.1007/978-3-642-16599-3_17.
[49] R. Pegoraro, R.B. Halima, K. Drira, K. Guennoun, J.M. Rosário, A framework for monitoring and runtime recovery of web service-based applications, in: J. Cordeiro, J. Filipe (Eds.), ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Vol. ISAS-2, Barcelona, Spain, June 12-16, 2008, 2008, pp. 201–206.
[50] S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data, IEEE Trans. Serv. Comput. 5 (2) (2012) 192–206, http://dx.doi.org/10.1109/TSC.2011.10.
[51] M.A. Köhl, H. Hermanns, S. Biewer, Efficient monitoring of real driving emissions, in: C. Colombo, M. Leucker (Eds.), Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, in: Vol. 11237 of Lecture Notes in Computer Science, Springer, 2018, pp. 299–315, http://dx.doi.org/10.1007/978-3-030-03769-7_17.
[52] S. Varvaressos, K. Lavoie, S. Gaboury, S. Hallé, Automated bug finding in video games: A case study for runtime monitoring, Comput. Entertain. 15 (1) (2017) 1:1–1:28, http://dx.doi.org/10.1145/2700529.
[53] P. Moosbrugger, K.Y. Rozier, J. Schumann, R2U2: Monitoring and diagnosis of security threats for unmanned aerial systems, Formal Methods Syst. Des. 51 (1) (2017) 31–61, http://dx.doi.org/10.1007/s10703-017-0275-x.
[54] S.D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S.A. Smolka, E. Zadok, Runtime verification with state estimation, in: Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers, 2011, pp. 193–207, http://dx.doi.org/10.1007/978-3-642-29860-8_15.
[55] K. Havelund, D.A. Peled, D. Ulus, The dejavu runtime verification benchmark, 2018.
[56] J. Vallet, A. Mrad, S. Hallé, É. Beaudet, The relational database engine: An efficient validator of temporal properties on event traces, in: E. Bagheri, D. Gasevic, S. Hallé, M. Hatala, H.R.M. Nezhad, M. Reichert (Eds.), 17th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOC Workshops, Vancouver, BC, Canada, September 9-13, 2013, IEEE Computer Society, 2013, pp. 275–284, http://dx.doi.org/10.1109/EDOCW.2013.37.
[57] A. Khalid, L.C. Briand, Checking data completeness in test data using runtime verification, in: Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, 2015, pp. 276–290, http://dx.doi.org/10.1007/978-3-662-46675-9_17.
[58] J. Piechotta, D. Holling, R. Hähnle, A. Podelski, Online detection of multiple violations in requirements specifications, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, the Netherlands, July 16-21, 2018, 2018, pp. 82–93, http://dx.doi.org/10.1145/3213846.3213872.
[59] V. Arora, F. van Breugel, P. Fischer, A. Gorbenko, Monitoring CSV data using multi-parametric run-time interval logic, in: IEEE International Conference on Software Engineering and Formal Methods, SEFM 2017, Trento, Italy, September 6-10, 2017, 2017, pp. 283–298, http://dx.doi.org/10.1007/978-3-319-66197-1_19.
[60] M. Lupp, Extensible markup language, in: S. Shekhar, H. Xiong, X. Zhou (Eds.), Encyclopedia of GIS, Springer, 2017, p. 583, http://dx.doi.org/10.1007/978-3-319-17885-1_400.
[61] S. Hallé, R. Villemaire, Runtime verification for the web - a tutorial introduction to interface contracts in web applications, 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 0 (2010) St Julians, Malta, November 1-4, 2010. Proceedings, in: Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 106–121, http://dx.doi.org/10.1007/978-3-642-16612-9_10.
[62] 1849-2016 - IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams, 2016.
[63] K. Havelund, G. Reger, G. Rosu, Runtime verification past experiences and future projections, in: B. Steffen, G.J. Woeginger (Eds.), Computing and Software Science - State of the Art and Perspectives, in: Vol. 10000 of Lecture Notes in Computer Science, Springer, 2019, pp. 532–562, http://dx.doi.org/10.1007/978-3-319-91908-9_25.
[64] L. Baresi, M. Cominetti, M. Rossi, Jrec: A framework for runtime monitoring of web services, in: Fourth International Conference on Service Oriented Computing, ICSOC'06, IEEE, 2006, pp. 479–488, http://dx.doi.org/10.1007/11948148_12.
[65] G. Bacci, M. Bartoletti, A.M. Moggi, E. Tuosto, Axml: A tool for runtime verification of xml documents, in: 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2011, Springer, 2011, pp. 228–232, http://dx.doi.org/10.1007/978-3-642-19835-9_18.
[66] C. Colombo, M. Pradella, M. Rossi, Xmonitor: A runtime verification tool for xml documents, in: 7th International Conference on Runtime Verification, RV 2017, Springer, 2017, pp. 226–233, http://dx.doi.org/10.1007/978-3-319-68167-2_15.
[67] D. Barrera, D. Perez-Palacin, R. Barrado, J. Calvo-Manzano, T. San Feliu, J. Garcia-Garcia, Flint: Fast log inspection for runtime verification of complex system interactions, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2019, pp. 447–457, http://dx.doi.org/10.1109/ICSME.2019.00058.

[68] M. Kowalski, S. Hoffmann, J. Halleux, Umbral: A stream processing language for runtime verification of real-time systems, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 688–699, http://dx.doi.org/10.1109/ASE.2019.00065.
[69] L. Moura, A. Sampaio, M. Souza, J.P. Feitosa, A. Oliveira, R. Marinho, Varan: A tool for runtime monitoring and verification of system software, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion, ICSE-Companion, IEEE, 2018, pp. 503–504, http://dx.doi.org/10.1145/3183440.3183460.
[70] L. Holík, M. Koreň, M. Novák, J. Šimáček, J. Třmač, Panda: Monitoring and diagnosis of distributed systems, arXiv preprint arXiv:1905.11953.
[71] M. Aghaei, L. Baresi, C. Ghezzi, Medusa: A runtime verification framework for data-centric applications, in: Proceedings of the 40th International Conference on Software Engineering, ICSE, ACM, 2018, pp. 89–99, http://dx.doi.org/10.1145/3180155.3180190.
[72] F. Chen, G. Rosu, Java-MOP: A monitoring oriented programming environment for Java, in: N. Halbwachs, L.D. Zuck (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings, in: Vol. 3440 of Lecture Notes in Computer Science, Springer, 2005, pp. 546–550, http://dx.doi.org/10.1007/978-3-540-31980-1_36.
[73] H. Garavel, R. Mateescu, SEQ.OPEN: A tool for efficient trace-based verification, in: S. Graf, L. Mounier (Eds.), Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004. Proceedings, in: Vol. 2989 of Lecture Notes in Computer Science, Springer, 2004, pp. 151–157, http://dx.doi.org/10.1007/978-3-540-24732-6_11.
[74] T. Agarwal, Finite state machine: Mealy state machine and moore state machine, 2020.
[75] F.A. Siddique, T.J.T. II, N. Brunelle, K. Skadron, Deterministic vs. non deterministic finite automata in automata processing, CoRR abs/2210.10077. http://dx.doi.org/10.48550/arXiv.2210.10077. arXiv:2210.10077.
[76] M.O. Rabin, Probabilistic automata, Inf. Control 6 (3) (1963) 230–245, http://dx.doi.org/10.1016/S0019-9958(63)90290-0.
[77] S. Konur, A survey on temporal logics for specifying and verifying real-time systems, Front. Comput. Sci. 7 (3) (2013) 370–403, http://dx.doi.org/10.1007/s11704-013-2195-2.
[78] R. Alur, D.L. Dill, A theory of timed automata, Theoret. Comput. Sci. 126 (2) (1994) 183–235, http://dx.doi.org/10.1016/0304-3975(94)90010-8.
[79] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.
[80] P. Cerný, T.A. Henzinger, A. Radhakrishna, Quantitative simulation games, in: Z. Manna, D.A. Peled (Eds.), Time for Verification, Essays in Memory of Amir Pnueli, in: Vol. 6200 of Lecture Notes in Computer Science, Springer, 2010, pp. 42–60, http://dx.doi.org/10.1007/978-3-642-13754-9_3.
[81] R. Khoury, S. Hallé, Tally keeping-LTL: An LTL semantics for quantitative evaluation of LTL specifications, in: 2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018, 2018, pp. 495–502, http://dx.doi.org/10.1109/IRI.2018.00079.
[82] R. Koymans, Specifying real-time properties with metric temporal logic, Real Time Syst. 2 (4) (1990) 255–299, http://dx.doi.org/10.1007/BF01995674.
[83] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, Z. Manna, LOLA: Runtime monitoring of synchronous systems, in: 12th International Symposium on Temporal Representation and Reasoning, TIME 2005, 23-25 2005, Burlington, Vermont, USA, IEEE Computer Society, 2005, pp. 166–174, http://dx.doi.org/10.1109/TIME.2005.26.
[84] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, D. Thoma, Tessla: Temporal stream-based specification language, in: T. Massoni, M.R. Mousavi (Eds.), Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018. Proceedings, in: Vol. 11254 of Lecture Notes in Computer Science, Springer, 2018, pp. 144–162, http://dx.doi.org/10.1007/978-3-030-03044-5_10.
[85] L. Franceschini, RML: Runtime Monitoring Language (Ph.D. thesis), University of Genoa, Italy, 2020, http://dx.doi.org/10.15167/franceschini-luca_phd2020-03-19, http://hdl.handle.net/11567/1001856.
[86] D. Ancona, L. Franceschini, A. Ferrando, V. Mascardi, RML: Theory and practice of a domain specific language for runtime verification, Sci. Comput. Program. 205 (2021) 102610, http://dx.doi.org/10.1016/j.scico.2021.102610.
[87] C. Colombo, G.J. Pace, G. Schneider, Dynamic event-based runtime monitoring of real-time and contextual properties, in: D.D. Cofer, A. Fantechi (Eds.), Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008. Revised Selected Papers, in: Vol. 5596 of Lecture Notes in Computer Science, Springer, 2008, pp. 135–149, http://dx.doi.org/10.1007/978-3-642-03240-0_13.
[88] C. Colombo, G.J. Pace, G. Schneider, LARVA — safer monitoring of real-time Java programs (tool paper), in: D.V. Hung, P. Krishnan (Eds.), Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27, 2009, IEEE Computer Society, 2009, pp. 33–37, http://dx.doi.org/10.1109/SEFM.2009.13.
[89] C. Colombo, G.J. Pace, Runtime verification using LARVA, 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, 3 of Kalpa Publications in Computing, EasyChair, 2017, pp. 55–63, http://dx.doi.org/10.29007/n7td.
[90] P.O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 249–289, http://dx.doi.org/10.1007/s10009-011-0198-6.
[91] G. Reger, H.C. Cruz, D.E. Rydeheard, Marq: Monitoring at runtime with QEA, in: C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, in: Vol. 9035 of Lecture Notes in Computer Science, Springer, 2015, pp. 596–610, http://dx.doi.org/10.1007/978-3-662-46681-0_55.
[92] H. Barringer, A. Goldberg, K. Havelund, K. Sen, 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, in: Vol. 2937 of Lecture Notes in Computer Science, Springer, 2004, pp. 44–57, http://dx.doi.org/10.1007/978-3-540-24622-0_5.
[93] A. Goldberg, K. Havelund, 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, 2005, INSTICC Press, 2005.
[94] I. Aktug, K. Naliuka, Conspec - A formal language for policy specification, Sci. Comput. Program. 74 (1–2) (2008) 2–12, http://dx.doi.org/10.1016/j.scico.2008.09.004.
[95] R.S. Sandhu, P. Samarati, Access control: Principle and practice, IEEE Commun. Mag. 32 (9) (1994) 40–48.
[96] M. Mammass, F. Ghadi, An overview on access control models, Int. J. Appl. Evol. Comput. 6 (4) (2015) 28–38, http://dx.doi.org/10.4018/IJAEC.2015100103.
[97] G. Ahn, Discretionary access control, in: L. Liu, M.T. Özsu (Eds.), Encyclopedia of Database Systems, Second Edition, Springer, 2018, http://dx.doi.org/10.1007/978-1-4614-8265-9_135.
[98] N. Li, Discretionary access control, in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 353–356, http://dx.doi.org/10.1007/978-1-4419-5906-5_798.
[99] S.D.C. di Vimercati, Discretionary access control policies (DAC), in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 356–358, http://dx.doi.org/10.1007/978-1-4419-5906-5_817.
[100] S.D.C. di Vimercati, P. Samarati, Mandatory access control policy (MAC), in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed, Springer, 2011, p. 758, http://dx.doi.org/10.1007/978-1-4419-5906-5_822.
[101] B.M. Thuraisingham, Mandatory access control, in: L. Liu, M.T. Özsu (Eds.), Encyclopedia of Database Systems, Second Ed., Springer, 2018, http://dx.doi.org/10.1007/978-1-4614-8265-9_214.
[102] S.J. Upadhyaya, Mandatory access control, in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 756–758, http://dx.doi.org/10.1007/978-1-4419-5906-5_784.
[103] Y. Zhang, J.B.D. Joshi, Role-based access control, in: L. Liu, M.T. Özsu (Eds.), Encyclopedia of Database Systems, Second Ed., Springer, 2018, http://dx.doi.org/10.1007/978-1-4614-8265-9_320.
[104] V. Alturi, D.F. Ferraiolo, Role-based access control, in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 1053–1055, http://dx.doi.org/10.1007/978-1-4419-5906-5_829.
[105] I. Clark, Role-based access control, in: R. Herold (Ed.), Encyclopedia of Information Assurance, Taylor & Francis, 2011, http://dx.doi.org/10.1081/E-EIA-120046311.
[106] A. Estes, Access control matrix, in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 12–13, http://dx.doi.org/10.1007/978-1-4419-5906-5_771.
[107] H.C.A. van Tilborg, S. Jajodia (Eds.), Rule-based access control, in: Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, p. 1072, http://dx.doi.org/10.1007/978-1-4419-5906-5_1312.
[108] V.C. Hu, D.R. Kuhn, D.F. Ferraiolo, J. Voas, Attribute-based access control, Computer 48 (2) (2015) 85–88.

[109] L. Bouganim, Y. Guo, Database encryption, in: H.C.A. van Tilborg, S. Jajodia (Eds.), Encyclopedia of Cryptography and Security, Second Ed., Springer, 2011, pp. 307–312, http://dx.doi.org/10.1007/978-1-4419-5906-5_677.
[110] G. Cormode, D. Srivastava, Anonymized data: Generation, models, usage, in: F. Li, M.M. Moro, S. Ghandeharizadeh, J.R. Haritsa, G. Weikum, M.J. Carey, F. Casati, E.Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, V.J. Tsotras (Eds.), Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010. Long Beach, California, USA, IEEE Computer Society, 2010, pp. 1211–1212, http://dx.doi.org/10.1109/ICDE.2010.5447721.
[111] J.F. Marques, J. Bernardino, Analysis of data anonymization techniques, in: D. Aveiro, J.L.G. Dietz, J. Filipe (Eds.), Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2020, Volume 2: KEOD, Budapest, Hungary, November 2-4, 2020. SCITEPRESS, 2020, pp. 235–241, http://dx.doi.org/10.5220/0010142302350241.
[112] R.L. Wilson, P.A. Rosen, Protecting data through perturbation techniques: The impact on knowledge discovery in databases, J. Database Manage. 14 (2) (2003) 14–26, http://dx.doi.org/10.4018/jdm.2003040102.
[113] K. Chen, L. Liu, Privacy preserving data classification with rotation perturbation, in: Proceedings of the 5th IEEE International Conference on Data Mining, ICDM 2005, 27-30 2005, Houston, Texas, USA, IEEE Computer Society, 2005, pp. 589–592, http://dx.doi.org/10.1109/ICDM.2005.121.
[114] K. Chen, L. Liu, Geometric data perturbation for privacy preserving outsourced data mining, Knowl. Inf. Syst. 29 (3) (2011) 657–695, http://dx.doi.org/10.1007/s10115-010-0362-4.
[115] T. Revathi, D. Ramaraj, Challenges and methods of data perturbation techniques, 2017.
[116] N. Patel, S. Patel, A study on data perturbation techniques in privacy preserving data mining, 2016.
[117] S.R.M. Oliveira, O.R. Zaïane, Privacy preserving clustering by data transformation, J. Inf. Data Manag. 1 (1) (2010) 37–52.
[118] N. Tatbul, Load shedding, in: L. Liu, M.T. Özsu (Eds.), Encyclopedia of Database Systems, Second Ed., Springer, 2018, http://dx.doi.org/10.1007/978-1-4614-8265-9_211.
[119] C. Olston, J. Jiang, J. Widom, Adaptive filters for continuous queries over distributed data streams, in: A.Y. Halevy, Z.G. Ives, A. Doan (Eds.), Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003., ACM, 2003, pp. 563–574, http://dx.doi.org/10.1145/872757.872825.
[120] N. Tatbul, U. Çetintemel, S.B. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in: J.C. Freytag, P.C. Lockemann, S. Abiteboul, M.J. Carey, P.G. Selinger, A. Heuer (Eds.), Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003. Morgan Kaufmann, 2003, pp. 309–320, http://dx.doi.org/10.1016/B978-012722442-8/50035-5.
[121] S. Mehta, V. Pandit, A survey on sampling techniques and applications, in: P.S. Kumar, S. Parthasarathy, S. Godbole (Eds.), Proceedings of the 16th International Conference on Management of Data, 2010, Allied Publishers, Nagpur, India, 2010, p. 11.
[122] B. Bonakdarpour, S. Navabpour, S. Fischmeister, Sampling-based runtime verification, in: M.J. Butler, W. Schulte (Eds.), FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings, in: Vol. 6664 of Lecture Notes in Computer Science, Springer, 2011, pp. 88–102, http://dx.doi.org/10.1007/978-3-642-21437-0_9.
[123] M. Arnold, M.T. Vechev, E. Yahav, QVM: An efficient runtime for detecting defects in deployed systems, ACM Trans. Softw. Eng. Methodol. 21 (1) (2011) 2:1–2:35, http://dx.doi.org/10.1145/2063239.2063241.
[124] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S.A. Smolka, S.D. Stoller, E. Zadok, Software monitoring with controllable overhead, Int. J. Softw. Tools Technol. Transf. 14 (3) (2012) 327–347, http://dx.doi.org/10.1007/s10009-010-0184-4.
[125] L. Fei, S.P. Midkiff, Artemis: Practical runtime monitoring of applications for execution anomalies, in: M.I. Schwartzbach, T. Ball (Eds.), Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006, ACM, 2006, pp. 84–95, http://dx.doi.org/10.1145/1133981.1133992.
[126] D. Yuan, S. Park, P. Huang, Y. Liu, M.M. Lee, X. Tang, Y. Zhou, S. Savage, Be conservative: Enhancing failure diagnosis with proactive logging, in: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, 2012, pp. 293–306.
[127] H. Li, W. Shang, A.E. Hassan, Which log level should developers choose for a new logging statement? Empir. Softw. Eng. 22 (4) (2017) 1684–1716, http://dx.doi.org/10.1007/s10664-016-9456-2.
[128] Hongbin Liu, Mingzhi Huang, Iman Janghorban, Payam Ghorbannezhad, Chang Kyoo Yoo, Faulty sensor detection, identification and reconstruction of indoor air quality measurements in a subway station, in: ICCAS 2011-2011 11th International Conference on Control, Automation and Systems, International Conference on Control, Automation and Systems, 2011, pp. 323–328.
[129] M. Tiger, F. Heintz, Internat. J. Approx. Reason. 119 (2020) 325–352.
[130] A. Francalanza, J.A. Pérez, C. Sánchez, Runtime verification for decentralised and distributed systems, in: E. Bartocci, Y. Falcone (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics, in: Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018, pp. 176–210, http://dx.doi.org/10.1007/978-3-319-75632-5_6.
[131] G. Audrito, F. Damiani, V. Stolz, G. Torta, M. Viroli, Distributed runtime verification by past-ctl and the field calculus, J. Syst. Softw. 187 (2022) 111251, http://dx.doi.org/10.1016/j.jss.2022.111251.
[132] G. Audrito, R. Casadei, F. Damiani, V. Stolz, M. Viroli, Adaptive distributed monitors of spatial properties for cyber–physical systems, J. Syst. Softw. 175 (2021) 110908, http://dx.doi.org/10.1016/j.jss.2021.110908.
[133] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification, Springer, 1992, http://dx.doi.org/10.1007/978-1-4612-0931-7.
[134] H. Kallwies, M. Leucker, C. Sánchez, T. Scheffel, Anticipatory recurrent monitoring with uncertainty and assumptions, in: T. Dang, V. Stolz (Eds.), Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022. Proceedings, in: Lecture Notes in Computer Science, vol. 13498, Springer, 2022, pp. 181–199, http://dx.doi.org/10.1007/978-3-031-17196-3_10.
[135] H. Kallwies, M. Leucker, C. Sánchez, Symbolic runtime verification for monitoring under uncertainties and assumptions, in: A. Bouajjani, L. Holík, Z. Wu (Eds.), Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022. Proceedings, in: 13505 of Lecture Notes in Computer Science, Springer, 2022, pp. 117–134, http://dx.doi.org/10.1007/978-3-031-19992-9_8.
[136] D. Basin, F. Klaedtke, S. Müller, E. Zălinescu, Monitoring metric first-order temporal properties, Vol. 62, no. 2. http://dx.doi.org/10.1145/2699444.
[137] K.G. Larsen, Proof systems for satisfiability in Hennessy-Milner logic with recursion, Theoret. Comput. Sci. 72 (2& 3) (1990) 265–288, http://dx.doi.org/10.1016/0304-3975(90)90038-J.
[138] J. Li, J. Lee, L. Liao, A novel algorithm for training hidden Markov models with positive and negative examples, in: T. Park, Y. Cho, X. Hu, I. Yoo, H.G. Woo, J. Wang, J.C. Facelli, S. Nam, M. Kang (Eds.), IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2020, Virtual Event, South Korea, December 16-19, 2020, IEEE, 2020, pp. 305–310, http://dx.doi.org/10.1109/BIBM49941.2020.9313477.
[139] L.R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, Proc. IEEE 77 (2) (1989) 257–286, http://dx.doi.org/10.1109/5.18626.
[140] A. Tavanaei, A.S. Maida, Training a hidden Markov model with a Bayesian spiking neural network, J. Signal Process. Syst. 90 (2) (2018) 211–220, http://dx.doi.org/10.1007/s11265-016-1153-2.
[141] H. Franco, A.J. Serralheiro, A new discriminative training algorithm for hidden Markov models, in: The First International Conference on Spoken Language Processing, ICSLP 1990, Kobe, Japan, November 18-22, 1990. ISCA, 1990.
[142] G. Zhou, C. Yang, P. Lu, X. Chen, Runtime verification in uncertain environment based on probabilistic model learning, Math. Biosci. Eng. 19 (12) (2022) 13607–13627, http://dx.doi.org/10.3934/mbe.2022635.
[143] C. Sammut, G.I. Webb (Eds.), Baum-Welch algorithm, in: Encyclopedia of Machine Learning and Data Mining, Springer, 2017, p. 99, http://dx.doi.org/10.1007/978-1-4899-7687-1_59.
[144] R. Taleb, S. Hallé, R. Khoury, A modular runtime enforcement model using multi-traces, in: E. Aïmeur, M. Laurent, R. Yaich, B. Dupont, J. García-Alfaro (Eds.), Foundations and Practice of Security - 14th International Symposium, FPS 2021, Paris, France, December 7-10, 2021. Revised Selected Papers, in: Vol. 13291 of Lecture Notes in Computer Science, Springer, 2021, pp. 283–302, http://dx.doi.org/10.1007/978-3-031-08147-7_19.
[145] R. Taleb, R. Khoury, S. Hallé, A modular pipeline for enforcement of security properties at runtime, Ann. Telecommun. (2023) http://dx.doi.org/10.1007/s12243-023-00952-z, in press.
[146] E. Bartocci, R. Grosu, A. Karmarkar, S.A. Smolka, S.D. Stoller, E. Zadok, J. Seyster, Adaptive runtime verification, in: S. Qadeer, S. Tasiran (Eds.), Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012. Revised Selected Papers, in: Vol. 7687 of Lecture Notes in Computer Science, Springer, 2012, pp. 168–182, http://dx.doi.org/10.1007/978-3-642-35632-2_18.
[147] N. Alechina, M. Dastani, B. Logan, Norm approximation for imperfect monitors, in: A.L.C. Bazzan, M.N. Huhns, A. Lomuscio, P. Scerri (Eds.), International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014. IFAAMAS/ACM, 2014, pp. 117–124.