A Taxonomy for Classifying Runtime Verification Tools

Yliès Falcone[1(✉)], Srđan Krstić[2(✉)], Giles Reger[3(✉)], and Dmitriy Traytel[2(✉)]

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr
[2] Institute of Information Security, Department of Computer Science, ETH Zürich, Zurich, Switzerland
{srdan.krstic,traytel}@inf.ethz.ch
[3] University of Manchester, Manchester, UK
giles.reger@manchester.ac.uk

Abstract. Over the last 15 years Runtime Verification (RV) has grown into a diverse and active field, which has stimulated the development of numerous theoretical frameworks and tools. Many of the tools are at first sight very different and challenging to compare. Yet, there are similarities. In this work, we classify RV tools within a high-level taxonomy of concepts. We first present this taxonomy and discuss the different dimensions. Then, we survey RV tools and classify them according to the taxonomy. This paper constitutes a snapshot of the current state of the art and enables a comparison of existing tools.

1 Introduction

Runtime Verification (RV) [7,28,29,38] (or runtime monitoring) is (broadly) the study of methods to analyze the dynamic behavior of computational systems. The most typical analysis is to check whether a given run of a system satisfies a given specification and it is this general setting (and its variants) that we consider in this paper. Whilst topics such as *specification mining* or *trace visualization* are generally considered to be within this broad field, we do not include them in our discussion.

This paper presents a taxonomy of RV frameworks and tools and uses this to classify 20 selected tools. This work is timely for a number of reasons. Firstly, after more than 15 years of maturing, the field has reached a point where such a general view is needed. The last significant attempt at a taxonomy was in 2004 [24] and had a distinctly different focus to our own. Secondly, a number

The authors warmly thank Martin Leucker for the early discussions on the taxonomy and mind map representation. This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology). In particular, the taxonomy and classification benefited from discussions within working groups one and two of this action. We would also like to acknowledge input from participants of Dagstuhl seminar 17462 [34].

C. Colombo and M. Leucker (Eds.): RV 2018, LNCS 11237, pp. 241–262, 2018.
https://doi.org/10.1007/978-3-030-03769-7_14

of activities, such as the runtime verification competitions [4,6,30,48], the RV-CuBES workshop [46,49], two schools dedicated to RV [16], and a COST action [1] (including the development of a tutorial book on the topic [5]), have put the development of runtime verification tools into focus.

Terminology. The field of RV is broad and the used terminology is not yet unified. For the sake of clarity, let us fix the following terms:

- *Monitored system.* The system consisting of software, hardware, or a combination of the two, that is being monitored. Its behavior is usually abstracted as a *trace* object.
- *Trace.* A finite sequence of observations that represents (or in some cases approximates) the behavior of interest in the monitor system. The process of extracting/recording the trace is usually referred to as *instrumentation*.
- *Property.* A partition of traces. This may simply be a separation of traces into two sets or a more refined classification of traces.
- *Specification.* A concrete description of a property using a well-defined formalism.
- *Monitor.* A runtime object that is used to check properties. The monitor will receive observations from the trace (usually incrementally) and may optionally send information back to the monitored system, or to some other source.
- *RV framework.* A collection of a specification formalism, monitoring algorithm(s) (for generating and executing monitors), and (optional) instrumentation techniques that allows for runtime verification.
- *RV tool.* A concrete instantiation of an RV framework.

Contributions and Structure. This paper has two main contributions:

- *The Taxonomy.* We present a detailed taxonomy that defines seven major concepts used to classify runtime verification approaches (Sect. 2). Each of these seven concepts are refined and explained, with areas of possible further refinement identified.
- *The Classification.* We take 20 runtime verification tools and classify them in our taxonomy (Sect. 3). Tools were taken from the recent runtime verification competitions and RV-CuBES workshop and therefore represent a recent and relevant snapshot.

We then discuss what we have learned from these two activities (Sect. 4) before concluding with some comments on how we see this work developing in the future (Sect. 5).

2 A Taxonomy of Runtime Verification

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

Fig. 1. Mindmap overviewing the taxonomy of Runtime Verification

Development Process. This taxonomy was developed in an iterative process alongside the classification presented in Sect. 3. The seven main conceptual areas were identified as an initial starting point and extended with established dichotomies (e.g., offline vs online). Sub-concepts were then added and refined based on the focused classification process and a wider survey of tools (involving over 50 tools, not described in this paper). We have attempted to ensure that the taxonomy remains as general and flexible as possible.

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

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

2.1 Specification

The specification part of the taxonomy is depicted in Fig. 2. A specification indicates the intended system behavior (property), that is *what one wants to check* on the system behavior. It is one of the main inputs of a runtime verification framework designed before running the system. A specification exists within the context of a general **system model** i.e., the abstraction of the system being specified. The main part of this model is the form of observations (traces) made about the system (see Sect. 2.5) but may include other contextual information. A specification itself can be either **implicit** or **explicit**.

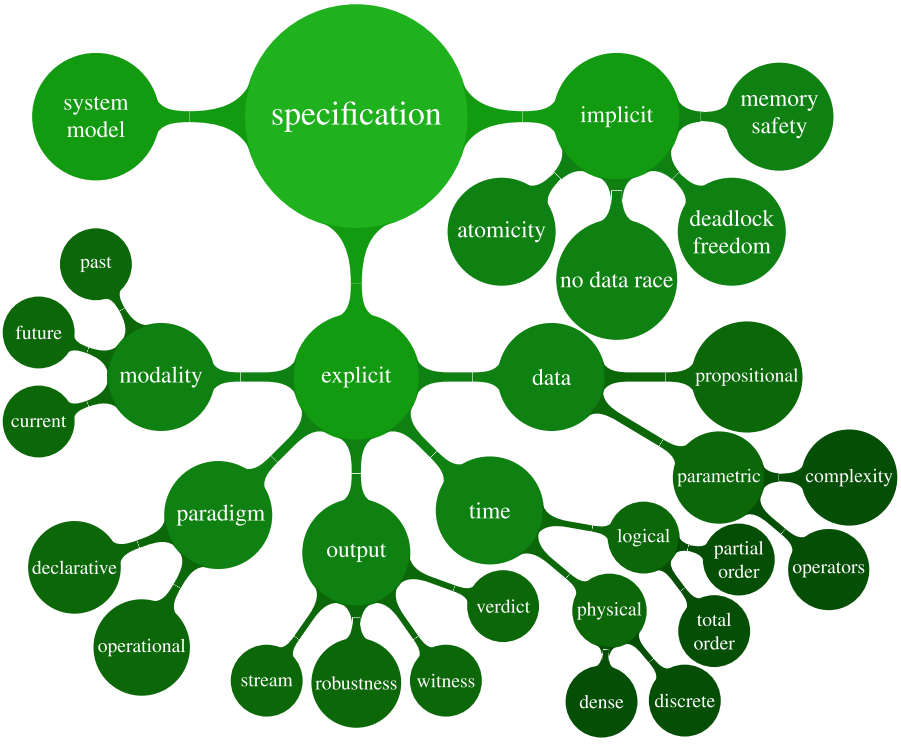

Fig. 2. Mindmap for the specification part of the taxonomy

Implicit Specifications. An **implicit** specification is used in a runtime verification framework when there is a general understanding of the particular desired behavior. Runtime verification tools do not require their users to explicitly formulate and enter implicit specifications. Implicit specifications generally aim at avoiding runtime errors (that would typically be not caught by a compiler). Such runtime errors can be critical. An example is memory safety, whose purpose is to ensure proper accesses to the system memory. Implicit specifications often

also describe correct concurrent behavior such as the absence of deadlocks, the atomicity of operations, and the absence of data races. The final layer here is a non-exhaustive list of prominent examples.

Explicit Specifications. An explicit specification is one provided by the user of the runtime verification framework and formally expresses functional or non-functional requirements. It can complement the properties checked by the compiler of a language (e.g., errors that would not be caught by type checking). An explicit specification denotes a function from traces to some **output** domain and is written in a specification formalism belonging to some **paradigm** e.g., specifications may describe this function **operationally** (e.g., by a finite-state automaton) or **declarative** (e.g., by a temporal logic formula). The specification formalism can offer different features used to model the expected behavior according to the dimensions discussed below.

The specification formalism may support different **modalities**. Some formalisms may restrict assertions to the **current** observation whereas others may support constraints over **past** or **future** observations. In some cases, different modalities represent distinct expressiveness classes; in other cases it is merely a matter of usability.

A key dimension is how specifications or a specification formalism handle data in observations. The simple case is the **propositional** case where observations are assumed to be atomic and unstructured (e.g., simple names). Otherwise, we say that the approach is **parametric**: observations (events or states) are associated with a list of (possibly named) runtime values. The structure of these runtime values may have different **complexity** e.g. they may be simple primitive values or complex XML documents or runtime objects. The **operators** over these values supported by the specification language may also vary. For example, whether it is possible to compare values in different ways (e.g. more than equality) or whether quantification (e.g. first-order, freeze quantification, or pseudo-quantification via templates) over parameters is supported [34–36].

A specification can also express constraints over **time**. Constraints can refer either to **logical time** or **physical time**. In the case of logical time, constraints are placed on the relative ordering between events. Such an order can be **total** (e.g., when monitoring a monolithic single-threaded program) or **partial** (e.g., when monitoring a multi-threaded program or a distributed system). In the case of physical time, timing constraints are related to the actual physical time that elapses when running the system. The domain of this timing information can be **discrete** or **dense**. There is a special case where *time is treated as data*. Such approaches typically do not offer native support for expressing quantitative temporal relationships, but use the parametrization operators to refer to timestamps.

The last dimension of an explicit specification formalism is that of the **outputs** assigned to the input executions e.g., the range of the denoted function. In the standard case, the specification associates **verdicts** with an execution. Those verdicts indicate specification fulfillment or violation and may range over a domain extending the Boolean domain. A more refined output might include

a **witness** for the verdict, e.g., a set of bindings of values to free variables in the specification that lead to violations. **Robustness** information extends classical verdicts by providing a quantitative assessment of the specification fulfillment or violation. Finally, in the most general case, specifications can describe output **streams**, which is any form of continuously produced information. This may be a stream of verdicts or witnesses, e.g., by evaluating the specification at each observation point, or more generally may be any data computed from the observations.

2.2 Monitor

The monitor part of the taxonomy is depicted in Fig. 3. A monitor is a main component of a runtime verification framework. By monitor, we refer to a component executed along the system for the purposes of the runtime verification process. A monitor implements a decision procedure which produces the expected output (either the related information for an implicit specification or the specification language output for an explicit specification).

The **decision procedure** of the monitor can be either **analytical** or **operational**. Analytical decision procedures query and scan records (e.g., from a

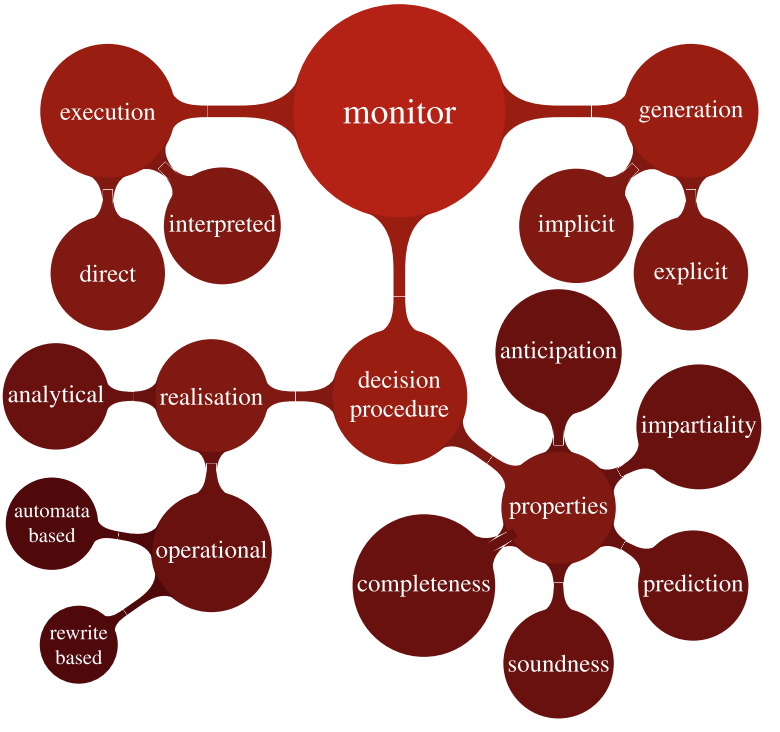

Fig. 3. Mindmap for the monitor part of the taxonomy

database) to determine whether some relations hold between the records and the current execution. Operational decision procedures are those based on automata or formula rewriting. In an **automata-based** monitor, the code relies on some form of automata-like formalism (either classical finite-state automata or richer forms). In a **rewrite-based** monitor, the decision procedure is based on a set of (possibly pre-defined) rewriting rules trigged by a new event. When designing monitors, it is desirable that its decision procedure guarantee several properties. Intuitively, a **sound** monitor never provides incorrect output, while a **complete** monitor always provide an output. The properties reflect how much confidence one can have in the output of monitor and how much confidence one can have that a monitor will produce an output, respectively. Soundness and completeness cannot be guaranteed in situations where, for instance, some form of sampling is used, not all necessary events can be observed by the monitor, or when the observation order does not correspond to the execution order. In such cases, the monitor can perform two kinds of **prediction**. Firstly, when the monitor produces its output as soon as possible, meaning that it uses a model of the monitored system to predict the possible futures of the trace and evaluate these possible futures before they actually happen. Secondly, when the monitor predicts potential errors in alternative concurrent executions (which are not actually observed by the monitor). A monitor is **impartial** when the produced outputs are not contradictory over time. Finally, a monitor can **anticipate** the output. This resembles prediction but the knowledge used by the monitor in this case comes from the monitored specification. Impartiality and anticipation are properties of the semantics of the specification language itself.

The decision procedure will act on an object (e.g. an automaton) which is itself often referred to as the monitor. This may be **generated explicitly** from the specification (e.g. an automaton synthesized from an LTL formula) or may exist **implicitly** (e.g. a rewrite system defined in an internal domain-specific language). Finally, a monitor must be **executed**. This might be **directly** if the monitor is given as code e.g., it is either already implemented as some extension of a programming language (i.e., an internal domain-specific language, or the synthesis step from generation directly produced executable code. Otherwise, the monitor is said to be **interpreted**. The key difference between the two approaches is whether each monitor is implemented by a different piece of code (direct) or there is a generic monitoring code that is parametrized by some monitor information (interpreted).

2.3 Deployment

The deployment part of the taxonomy is depicted in Fig. 4. By deployment, we refer to how the monitor is effectively implemented, organized, how it retrieves the information from the system, and when it does so.

The notion of **stage** describes *when* the monitor operates, with respect to the execution of the system. Runtime verification is said to apply **offline** when the monitor runs after the system finished executing and thus has access to the complete system execution (e.g., a log file). It is said to apply **online** when the

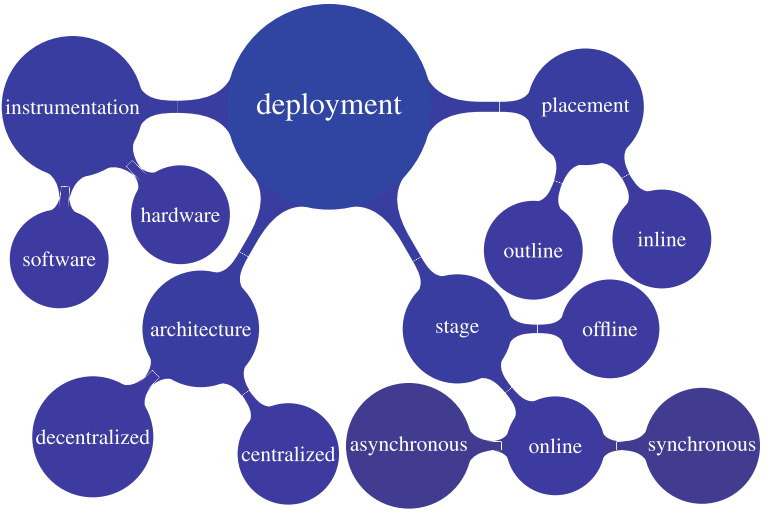

Fig. 4. Mindmap for the deployment part of the taxonomy

monitor runs while the system executes and thus observes the current execution and a part of its history. In the online case, the communication and connection between the monitor and the system can be **synchronous** or **asynchronous**, respectively depending on whether the initial program code stops executing when the monitor analyzes the retrieved information. It is possible for a monitor to be *partially* synchronous if it synchronises on some but not all observations.

The notion of **placement** describes where the monitor operates, with respect to the running system. Therefore, this concept only applies when the stage is online. Traditionally, the monitor is said to be **inline** (resp. **outline**) when it executes in the same (resp. in a different) address space as/than the running system. Pragmatically, the difference between inline and outline is a matter of **instrumentation**. An inline tool implicitly includes some form of instrumentation, used to inline the monitor in the monitored system. Conversely, outline tools typically provide an interface for receiving observations. This interface may exist within the same language and be called directly, or it may be completely separate with communication happening via other means (e.g., pipes). There is a (not uncommon) grey area between the two in the instance of tools that provide an outline interface but may also automatically generate instrumentation code. Instrumentation itself may be at the **hardware** or **software** level and there are further subdivisions within this that we do not cover here. Finally, the architecture of the monitor may be **centralized** (e.g., in one monolithic procedure) or **decentralized** (e.g., by utilising communicating monitors, which may be synchronous or asynchronous).

2.4 Reaction

The reaction part of the taxonomy is depicted in Fig. 5. By reaction, we refer to how the monitor affects the execution of the system; this can be passive or active.

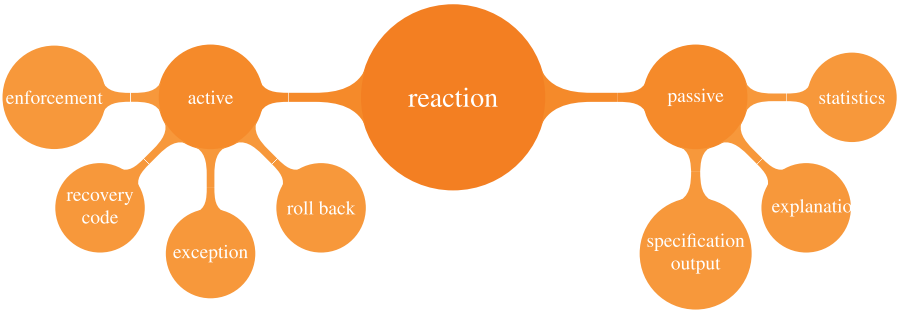

Fig. 5. Mindmap for the reaction part of the taxonomy

Reaction is said to be **passive** when the monitor does not influence or minimally influences the initial execution of the program. A passive monitor is typically an observer, only collecting information. This means that there are some sorts of guarantees that the analysis performed by monitor did not alter the execution and that the reported information is accurate. Such guarantees could be some form of behavioral equivalence (e.g., simulation, bisimulation, or weak bisimulation) between the initial system and the monitored system. In that case, the purpose of monitoring typically pertains to producing the **specification outputs** (e.g., verdicts or robustness information) or providing a form of **explanation** of a specification output (e.g., a witness trace containing the important events leading to a specific verdict) or **statistics** (for instance violated/satisfied specifications, number of times intermediate verdicts were output before a final verdict is reached).

Reaction is said to be **active** when the monitor affects the execution of the monitored system. An active monitor would typically affect the execution of the system when a violation is reported or detected to be irremediably happening. Various interventions are possible. A so-called **enforcement** monitor can try to prevent property violations from occurring by forcing the system to adhere to the specification. When a violation occurs, a monitor can execute **recovery code** to mitigate the effect of the fault and let the program either terminate or pursue the execution from a safer state. A monitor can also raise **exceptions** that were already present in the initial system. Finally, recovery mechanisms can be launched to **roll the system back** in a previous correct state.

2.5 Trace

The trace part of the taxonomy is depicted in Fig. 6. The notion of trace appears in two places in a runtime verification framework and this distinction is captured by the **role** concept. By **observed** trace we refer to the object extracted from the monitored system and examined by the monitor. Conversely the trace **model** is the mathematical object forming part of the semantics of the specification formalism. Clearly, a monitoring approach must connect the two but it can be important to be clear about what properties they have separately. For example, trace models may be **infinite** (as in standard LTL) whilst observed traces are necessarily **finite** – in such case the monitoring approach must evaluate a finite trace with respect to a property over infinite traces. A trace model must reflect the notions of time and data present in the specification (see Sect. 2.1).

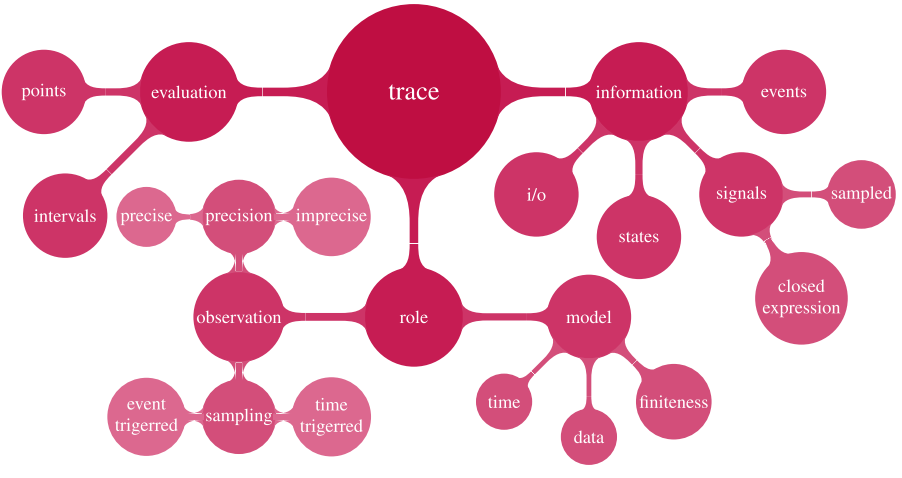

Fig. 6. Mindmap for the trace part of the taxonomy

The construction of the observed trace is also parameterized by a **sampling** decision and a **precision** decision. Sampling is said to be **event triggered** when the monitor gets information from the target system when an event of interest happens in the program. This can be the case when an event occurs in the system (in case the trace consists of a collection of events), when a relevant part of the program state changes, or when a new input is received or an output produced by the system. Sampling is said to be **time triggered** when there exists a more or less regular period at which information is collected on the program. The term sampling here reflects the fact that any trace will only collect a relevant subset of actual behaviours. If the trace contains *all* relevant traces then it is **precise**, otherwise it is **imprecise**. Reasons for imprecision might be imperfect trace collection methods, or purposefully for reasons of overhead.

Either form of trace object is an abstraction of the system execution and only contains some of the runtime information. The **information** retrieved from the program can take the form of isolated information. For instance, the trace can contain information on the internal **state** of the program or notifications that some **events** occurred in the program (or both). Not exclusive of the previous option, the monitor can also process the **input and output** information from a transformational program. Finally, the analyzed object can consist in time-continuous information in the form of a **signal**, which may be captured as a **closed-expression** or by discrete **sampling**.

The runtime information retrieved by the monitor represents an **evaluation** of the system state. This information can be related to an identified **point** (in time or at a program location) or an **interval**.

2.6 Interference

The **interference** part of the taxonomy (see Fig. 1) characterizes monitoring frameworks as **invasive** or **non-invasive**. In absolute, a non-invasive monitoring framework being impossible (observer effect), this duality corresponds more in reality to a spectrum. There are two sources of interference for a monitoring framework with a system. The interference with the system execution can be for instance related to the induced overhead (time and memory wise) or by a modification of scheduling. First, how much a runtime verification framework interferes with the initial system depends on the effect of the instrumentation applied to the system, which itself depends on the specification as instrumentation is purposed mainly to collect a trace. Thus, the quantity of information in the trace and the frequency at which this information is collected (depending on the sampling) affects the instrumentation. Moreover, interference also depends on the monitor deployment. Offline monitoring is considered to be less intrusive because the observation made on the system consists only in dumping events to a trace. Online monitoring is considered to be more intrusive to a degree depending on the coupling between the system and the monitor. Second, interference with the monitored system also occurs when actively steering the system.

2.7 Application Areas

We have included **application areas** as a top-level concept of the taxonomy (see Fig. 1) as it can have a large impact on other aspects of the runtime verification tools. There are numerous application areas of runtime verification. We have identified the following (certainly non-exhaustive) categories. First, runtime verification can be used for the purpose of **collecting information** on a system. This includes visualizing the execution (e.g., with traces, graphs, or diagrams) and evaluating the runtime performance (in a broad sense) over some metrics (execution time, memory consumption, communication, etc.) to collect statistics. Second, runtime verification can be used to perform an **analysis** of the system, usually to complement or in conjunction with static analysis techniques. This could focus on assessing concerns for a system in a large sense

(e.g., requirements, properties, or goals) of **security and privacy**, **safety** and **progress/liveness** natures. Third, runtime verification can be used to augment software engineering techniques with a rigorous analysis of runtime information. Fourth, runtime verification can be used to complement techniques for finding defects and locating faults in systems such as **testing** and **debugging**. Finally, leveraging the previous techniques, runtime verification can be used to address the general problem of runtime **failure prevention and reaction**, by offering ways to detect faults, contain them, recover from them, and repair the system.

3 Classification

This section considers a snapshot of runtime verification tools with respect to the previously introduced taxonomy. As discussed later, this is the first major step in our effort to achieve a full classification of all existing runtime verification tools.

Tool Selection. For our initial classification we wanted a reasonably sized set that represented reasonably active, well documented, and recent tools. In particular, a recent source of information about the tool was of utmost importance. We therefore focussed on the entrants to the runtime verification competitions taking place between 2014 and 2016 [6,30,48] and the submissions to the RV-CuBES workshop [49], which took place in 2017. This led to a selection of 20 tools (14 from the competition and 6 from the workshop). Our selection method is biased as the competition focussed only on tools for Java, C, or offline monitoring. Therefore, we expect our initial classification to be biased towards this area of the taxonomy. The selection is also favoring tools that had the resources to participate in either the competition or workshop.

Participating Tools. The 20 participating tools are listed in Table 1 along with hyperlinks (where applicable), references, the name of their specification formalism and some additional remarks. The given references are those that were used to fill in the classification, along with any additional information provided in competition reports [6,30,48].

Classification. The classification is given in Table 3 with a legend found in Table 2. We leave a general discussion of the classification to the next section. The classification also exists as a living document found at https://goo.gl/Mmuhdd. This document welcomes comments from the community and will be updated as our work continues.

The classification non-uniformly instantiates levels for different parts of the taxonomy. We omit parts of the taxonomy that are too abstract to be properly instantiated for the participating tools (e.g., system model) or if the source material of all the tools does not contain enough information for classification. Moreover, the major concepts application area and interference are omitted since there is a large space of non-exclusive possibilities that each tool can instantiate in this part of the taxonomy.

Table 1. Details of the participating tools.

Tool	References	Specification formalism name + some remarks
Aerial	[8,12,13]	MDL
ARTiMon	[44]	ARTiMon (no quantification; only aggregation)
BeepBeep	[31,32]	Stateful stream function API + DSLs for LTL-FO+, FSM
DANA	[27]	UML state machines
detectEr	[15]	μHML (only universal quantification)
E-ACSL	[23,53]	ACSL/implicit
JavaMOP	[37,40]	MOP with plugins (LTL, FMS, ERE, CFG, SRS, CaReT)
jUnitRV	[21,22]	Temporal Data Logic
Larva	[17–19]	DATEs
LogFire	[33]	LogFire DSL
MarQ/QEA	[3,45,47]	QEA
MonPoly	[9–11]	MFOTL
Mufin	[20]	Projection Automata
R2U2	[42,50,51]	MTL + mission time LTL
RiTHM	[14,43]	LTL_3
RTC	[41]	- (implicit)
RV-Monitor	[39]	MOP (see above)
STePr	-	Scala-internal DSL
TemPsy/OCLR-Check	[25,26]	TemPsy
VALOUR	[2]	Valour Script/Rules

The classification also refines the taxonomy. For instance, software instrumentation is refined based on how it is implemented (using AspectJ or reflection). We also provide more detailed description of the decision procedure, whenever the tools' material provides such information. Besides the values specified in Table 2, the cells in Table 3 may contain values "all", or "none" indicating that the tool supports all, or none of the features defined by that part of the taxonomy. Value "na" states that this part of the taxonomy is not applicable to the tool, while "?" means that there is insufficient information about the tool to establish a definitive classification. We have devised the classification mostly without involvement of the tools' developers, based on the available materials. As future work, we will validate our understanding of the tools through targeted interviews whenever this will be possible. (Any corrections will appear in the online version of the classification.)

Threats to Validity. Whilst we argue that this classification represents a reasonable snapshot of current runtime verification tools, there are two possible threats to its validity. Firstly, the sample of tools is heavily focussed towards software-monitoring with explicit specifications. Although, within this focus the coverage of tools is broad. It is important to be clear about the *scope* of this work. Few tools supporting implicit specifications (e.g. MemorySanitizer [54] or ThreadSanitizer [52]) identify themselves as runtime verification tools and most existing work does not share much of the terminology with runtime verification e.g. it is not usual to abstract a system by a trace. Whilst such tools can be categorised in the taxonomy, their classification will remain coarse as such tools are not the focus of the taxonomy. In general, this suggests that some areas of the taxonomy may require a refinement in the future, but also that these refinements will be orthogonal to the work presented here. We discuss this further later.

Secondly, the classification does not cover all known tools (over 50). However, many tools not included in this classification are mostly of historic interest. Others have influenced the taxonomy without taking part in the classification (e.g. stream-based approaches). Nonetheless, it will be important to achieve maximal coverage in the future.

4 Discussion

This section makes some observations about the taxonomy and classification.

4.1 General Observations

The majority of tools handled explicit specifications based on totally ordered logical time. There was a mixture of propositional and parametric tools and different approaches to physical time. Almost all tools were event-based with event-triggered sampling.

Unsurprisingly, the monitor decision procedures were varied, with many not quite fitting the mould. The majority of tools were online – it is perhaps worth observing that RV-Monitor added an offline interface for the competition. Only one tool is purely offline. The distinction between operational and declarative specification languages results in two sets of tools of roughly the same size. Both approaches are useful and favored by different sub-communities of RV. A few tools support both kinds of specification languages.

Some parts of the taxonomy were relatively straightforward to complete, whereas others were more controversial. The most discussed part of the taxonomy was the monitor concept as the term “monitor” is highly overloaded in our field and many frameworks do not have an explicit notion of a monitor. In the end, we decided to split how a monitor is generated and how it is executed as there is not necessarily a close link between the two. Another area that was difficult to fix was the relation between trace model and observed trace. It would be wrong to conflate the two, however often these concepts overlap.

Table 2. Key for Table 3.

Column	Values
Specification	
implicit	ms = *memory safety*
data	p = *propositional*, s = *simple parametric*, c = *complex parametric*
output	s = *stream*, v = *verdict*, w = *witness*, r = *robustness*
logical time	tot = *total order*, par = *partial order*
physical time	$\mathbb{N}$ = *discrete*, $\mathbb{R}$ = *dense*, none = *no time*
modality	f = *future*, p = *past*, c = *current*
Monitor	
generation	e = *explicit*, i = *implicit*
execution	i = *interpreted*, d = *direct*
Deployment	
stage	on = *online*, off = *offline*
synchronisation	sync = *synchronous*, async = *asynchronous*
architecture	c = *centralised*, d = *decentralised*
placement	out = *outline*, in = *inline*
instrumentation	sw = *software*, swAJ = *software with AspectJ*, swR = *software with reflection*
Reaction	
active	e = *exception*, r = *recovery*
passive	so = *specification output*, e = *explanations*
Trace	
information	e = *events*, s = *states*
sampling	et = *event-triggered*, tt = *time-triggered*
evaluation	p = *points*, i = *intervals*
precision	p = *precise*, i = *imprecise*
model	f = *finite trace model*, i = *infinite trace model*
General	
	all = *all features supported*, none = *no features supported*
	na = *not applicable*, ? = *insufficient information*

4.2 Underpopulated Areas of the Taxonomy

The classification unveils areas that are not populated by any tools. We discuss the main ones here and what this might mean.

Decentralized Architecture. This appears to be an area that has not received much attention. This may be due to the inherent complexity of decentralization, or it may reflect a lack of need. They may also be interdependencies with the

Table 3. Classification of participating tools.

Tool	Specification							Monitor			Deployment					Reaction		Trace				
	implicit	explicit				modality	paradigm	decision procedure	generation	execution	stage	synchronisation	architecture	placement	instrumentation	active	passive	information	sampling	evaluation	precision	model
		data	output	time																		
				logical	physical																	
Aerial	none	p	s	tot	$\mathbb{N}$	all	d	dynamic programming	i	i	on	none	c	out	none	none	so	e	et	p	p	i
ARTiMon	none	s	s	tot	$\mathbb{NR}$	all	d	?	i	i	on	none	c	out	none	none	so	e	et	i	p	i
BeepBeep	none	c	s	tot	none	f	all	stream-processing	i	d	on	all	c	out	sw	none	so	e	et	p	p	i
DANA	none	p	s	tot	$\mathbb{R}$	all	o	?	i	d	on	sync	c	?	?	none	so	e	et	p	p	f
detectEr	none	s	v	par	none	f	d	dynamic programming	e	i	on	all	c	in	sw	none	so	e	et	p	p	i
E-ACSL	ms	na	r	?	na	na	o	code rewriting with assertions	e	d	on	sync	c	in	sw	e	e	s	na	na	na	na
JavaMOP	none	s	w	tot	none	all	all	trace slicing plugin-based	e	d	on	sync	c	in	swAJ	r	e	e	et	p	p	f
jUnitRV	none	s	v	tot	none	f	d	automata-based (modulo theories e.g. SMT solver)	e	d	on	sync	c	in	swR	?	?	e	et	p	p	f
Larva	none	s	v	tot	$\mathbb{N}$	f	o	automata-based	e	d	on	all	c	all	sw	r	so	e	et	p	p	f
LogFire	none	s	w	tot	none	all	o	rewriting-based (RETE)	i	d	all	sync	c	out	none	none	so	e	et	p	p	f
MarQ/QEA	none	s	v	tot	$\mathbb{N}$	f	o	automata-based	i	d	all	sync	c	all	sw	none	so	e	et	p	p	f
MonPoly	none	s	s	tot	$\mathbb{N}$	all	d	first-order queries	i	i	on	none	c	out	none	none	so	e	et	p	p	all
Mufin	none	s	v	tot	none	f	o	automata-based (union-find)	i	d	on	sync	c	out	none	none	so	e	et	p	p	f
R2U2	none	p	s	tot	$\mathbb{N}$	all	d	automata-based	e	i	on	async	c	out	none	none	so	e	et	p	p	i
RiTHM	none	p	s	tot	none	f	o	time-triggered runtime verification	e	d	on	async	c	in	sw	none	so	s	all	p	p	i
RTC	ms	na	w	?	na	na	o	?	i	d	on	sync	c	in	sw	r	?	?	et	p	p	na
RV-Monitor	none	s	w	tot	$\mathbb{N}$	all	all	(see JavaMOP)	i	d	all	sync	c	all	sw	r	e	e	et	p	p	f
STePr	none	s	s	tot	$\mathbb{N}$	all	o	?	i	d	on	?	c	out	none	none	so	e	et	p	p	?
TemPsy/OCLR-Check	none	p	v	tot	$\mathbb{N}$	all	d	OCL constraint	i	i	off	na	c	out	none	none	so	e	et	p	p	f
VALOUR	none	s	v	tot	$\mathbb{N}$	all	o	automata-based	i	d	on	all	c	in	swAJ	none	all	e	all	p	p	f

monitoring setting (e.g. the language of interest) that make such an approach less desirable. Also the selection of the tools based on the competitions might have contributed to this topic being underrepresented: the competitions did not focus on the distributed setting.

Monitoring States. None of the tools in our classification monitor states of a program directly. This may be a result of the popularity of event-oriented specification languages. This is an interesting observation as it is commonly stated as a common dichotomy (observing events or states) but we do not see this in our classification. Although, arguably E-ACSL monitors states even though it has no formal notion of a trace. Furthermore, the distinction between state and event is not always clear; it is always possible to encode state in events and some inline tools allow specifications to directly query runtime state.

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

Applications. Many of the tools were not developed with a single application area in mind, making this part of the taxonomy irrelevant. However, in cases where an application exists it is significant. For example, R2U2 is designed to monitor unmanned aerial vehicles and this is heavily reflected in the tool's design. This is less an underpopulated area and more an area that only applies in certain cases.

4.3 Relation to Other Classifications

We briefly compare our taxonomy to the previous most complete taxonomy for runtime monitoring [24]. The context of this taxonomy is slightly different as their focus was software-fault monitoring. We have chosen to focus more on issues related to the monitoring of *explicit specifications* and include fewer operational issues. Delgado et al. identify four top-level concepts: *Specification*, *Monitor*, *Event-Handler*, and *Operational Issues*. Below we summarise the most significant differences in each area.

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

Monitor. Again there is a focus on instrumentation, which is something that we do not consider in depth as we tend to draw a line between instrumentation and monitoring. They differentiate whether instrumentation is manual or automatic. They key observation here is that they view *placement* slightly differently, as they

classify monitoring occurring using different resources (e.g. running in a different process) as *offline*. We refer to [7] for a discussion on the recent alternatives when considering instrumentation.

Event-Handler. This concept has the same meaning as our concept of *reaction* and their sub-concepts are subsumed by ours.

Operational Issues. This is a concept that we have not considered in our taxonomy. They focus on *source program type* i.e. the types of programs that it can work with (e.g. just Java), *dependencies* (e.g. on specific hardware), and *maturity* of the tool. This is something we could extend our taxonomy with but we found that many tools are actively developed and such data may quickly become outdated.

5 Conclusion

We have introduced a taxonomy for classifying runtime verification tools and used it to classify an initial set of 20 tools taken from recent competitions and workshops. We believe that this classification activity is important for a number of reasons. Firstly, the taxonomy fixes shared terminology and dimensions for discussing tools – it is important that the community has a shared language for what it does. Secondly, the classification exercise gives an overview of comparable tools, making it more straightforward to identify the tools against which new contributions should be compared. Additionally, the taxonomy can help shape evaluation and benchmark activities in general, in particular the design of competitions. Finally, we believe this kind of activity can identify interesting directions for future research, in particular in underpopulated areas of the taxonomy.

Our work is ongoing and our next step is to extend the classification. We have collected information about over 50 runtime verification tools and plan to extend the classification to these tools. This may constitute a challenge because many of the tools from this extended list do not provide sufficient information for classification.

References

1. IC1402 Runtime Verification beyond Monitoring (ARVI). https://www.cost-arvi.eu/
2. Azzopardi, S., Colombo, C., Ebejer, J.P., Mallia, E., Pace, G.: Runtime verification using VALOUR. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 10–18. EasyChair (2017)
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9

4. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 1–9. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_1
5. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
6. Bartocci, E., et al.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. STTT, 1–40 (2017)
7. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
8. Basin, D.A., Bhatt, B.N., Traytel, D.: Almost event-rate independent monitoring of metric temporal logic. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 94–112. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_6
9. Basin, D.A., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: monitoring usage-control policies. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 360–364. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_27
10. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
11. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
12. Basin, D.A., Krstić, S., Traytel, D.: Almost event-rate independent monitoring of metric dynamic logic. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 85–102. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_6
13. Basin, D.A., Krstić, S., Traytel, D.: AERIAL: almost event-rate independent algorithms for monitoring metric regular properties. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 29–36. EasyChair (2017)
14. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Time-triggered runtime verification. Form. Methods Syst. Des. **43**(1), 29–60 (2013)
15. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A suite of monitoring tools for Erlang. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 41–47. EasyChair (2017)
16. Colombo, C., Falcone, Y.: First international summer school on runtime verification. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 17–20. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_2
17. Colombo, C., Pace, G.J.: Runtime verification using LARVA. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 55–63. EasyChair (2017)
18. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03240-0_13
19. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: Hung, D.V., Krishnan, P. (eds.) SEFM 2009, pp. 33–37. IEEE Computer Society (2009)

20. Decker, N., Harder, J., Scheffel, T., Schmitz, M., Thoma, D.: Runtime monitoring with union-find structures. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 868–884. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_54
21. Decker, N., Leucker, M., Thoma, D.: jUnit^{RV}–adding runtime verification to jUnit. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 459–464. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_34
22. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. STTT **18**(2), 205–225 (2016)
23. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Shin, S.Y., Maldonado, J.C. (eds.) SAC 2013, pp. 1230–1235. ACM (2013)
24. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004)
25. Dou, W., Bianculli, D., Briand, L.: A model-driven approach to offline trace checking of temporal properties with OCL. Technical report SnT-TR-2014-5, Interdisciplinary Centre for Security, Reliability and Trust (2014). http://hdl.handle.net/10993/16112
26. Dou, W., Bianculli, D., Briand, L.: TemPsy-Check: a tool for model-driven trace checking of pattern-based temporal properties. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 64–70. EasyChair (2017)
27. Drabek, C., Weiss, G.: DANA - description and analysis of networked applications. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 71–80. EasyChair (2017)
28. Falcone, Y.: You should better enforce than verify. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 89–105. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_9
29. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO SPS D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press, Amsterdam (2013)
30. Falcone, Y., Ničković, D., Reger, G., Thoma, D.: Second international competition on runtime verification. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 405–422. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_27
31. Hallé, S.: When RV meets CEP. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 68–91. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_6
32. Hallé, S., Khoury, R.: Event stream processing with BeepBeep 3. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 81–88. EasyChair (2017)
33. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015)
34. Havelund, K., Leucker, M., Reger, G., Stolz, V.: A shared challenge in behavioural specification (Dagstuhl seminar 17462). Dagstuhl Rep. **7**(11), 59–85 (2017)
35. Havelund, K., Reger, G.: Runtime verification logics a language design perspective. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 310–338. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_16

36. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
37. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: efficient parametric runtime monitoring framework. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE 2012, pp. 1427–1430. IEEE Computer Society (2012)
38. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
39. Luo, Q., et al.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_24
40. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012)
41. Milewicz, R., Vanka, R., Tuck, J., Quinlan, D., Pirkelbauer, P.: Lightweight runtime checking of C programs with RTC. Comput. Lang. Syst. Str. **45**, 191–203 (2016)
42. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Form. Methods Syst. Des. **51**(1), 31–61 (2017)
43. Navabpour, S., et al.: RiTHM: a tool for enabling time-triggered runtime verification for C programs. In: Meyer, B., Baresi, L., Mezini, M. (eds.) ESEC/FSE 2013, pp. 603–606. ACM (2013)
44. Rapin, N.: ARTiMon monitoring tool, the time domains. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 106–122. EasyChair (2017)
45. Reger, G.: An overview of MarQ. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 498–503. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_34
46. Reger, G.: A report of RV-CuBES 2017. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 1–9. EasyChair (2017)
47. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
48. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 21–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_3
49. Reger, G., Havelund, K. (eds.): RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, Kalpa Publications in Computing, vol. 3. EasyChair (2017)
50. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 357–372. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_24
51. Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime analysis with R2U2: a tool exhibition report. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 504–509. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_35

52. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA 2009, pp. 62–71. ACM, New York (2009). https://doi.org/10.1145/1791194.1791203, http://doi.acm.org/10.1145/1791194.1791203
53. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 164–173. EasyChair (2017)
54. Stepanov, E., Serebryany, K.: MemorySanitizer: fast detector of uninitialized memory use in C++. In: Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Francisco, CA, USA, pp. 46–55 (2015)