COST Action IC 1402 ArVI:
Runtime Verification Beyond Monitoring

Activity Report of Working Group 1
-
Core Aspects of Monitoring

Wolfgang Ahrendt[1], Cyrille Artho[2], Christian Colombo[3], Yliès Falcone[4],
Srdan Krstic[5], Martin Leucker[6], Florian Lorber[7], Joao Lourenço[8], Leonardo Mariani[9],
César Sánchez[10], Gerardo Schneider[11], and Volker Stolz[12]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] KTH Royal Institute of Technology, Stockholm, Sweden
[3] University of Malta, Msida, Malta
[4] Univ. Grenoble Alpes, CNRS, Inria, LIG, 38000 Grenoble, France
[5] Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
[6] University of Lübeck, Lübeck, Germany
[7] Department of Computer Science, Aalborg University, Aalborg, Denmark
[8] NOVA LINCS, DI, FCT, NOVA University Lisbon, Lisbon, Portugal
[9] University of Milano - Bicocca, IT-20126 Milan, Italy
[10] IMDEA Software Institute, Madrid, Spain
[11] University of Gothenburg, Gothenburg, Sweden
[12] Western Norway University of Applied Science, Bergen, Norway

Abstract. This report presents the activities of the first working group of the COST Action ArVI, Runtime Verification beyond Monitoring. The report aims to provide an overview of some of the major core aspects involved in Runtime Verification. Runtime Verification is the field of research dedicated to the analysis of system executions. It is often seen as a discipline that studies how a system run satisfies or violates correctness properties.

The report exposes a taxonomy of Runtime Verification (RV) presenting the terminology involved with the main concepts of the field. The report also develops the concept of instrumentation, the various ways to instrument systems, and the fundamental role of instrumentation in designing an RV framework. We also discuss how RV interplays with other verification techniques such as model-checking, deductive verification, model learning, testing, and runtime assertion checking. Finally, we propose challenges in monitoring quantitative and statistical data beyond detecting property violation.

1 Introduction

Runtime Verification (RV), as a field of research, is referred to by many names such as runtime monitoring, trace analysis, dynamic analysis, passive testing, runtime enforcement etc. (see [40,55,33,14,29,35] for tutorials). The term *verification* implies a notion of *correctness* with respect to some property. This is somewhat different from the term *monitoring* (the other popular term) which only suggests that there is some form of behaviour being observed. Some view the notion of monitoring as being more specific than that of verification as they take it to imply some interaction with the system, whereas verification is passive in nature.

RV is a lightweight, yet rigorous, formal method that complements classical exhaustive verification techniques (such as model checking and theorem proving) with a more practical approach that analyses a single execution trace of a system. At the expense of a limited execution coverage, RV can give very precise information on the runtime behaviour of the monitored system. The system considered can be a software system, hardware or cyber-physical system, a sensor network, or any system in general whose dynamic behaviour can be observed. The archetypal analysis that can be performed on runtime behaviour is to check for correctness of that behaviour. However, there are many other analyses (e.g., falsification analysis) or activities (e.g., runtime enforcement) that can be performed, as it will be discussed elsewhere in this report. RV is now widely employed in both academia and industry both before system deployment, for testing, verification, and debugging purposes, and after deployment to ensure reliability, safety, robustness and security.

The RV field as a self-named community grew out of the RV workshop established in 2001, which became a conference in 2010 and occurs each year since then. In 2014, we have initiated the international Competition on (Software for) Runtime Verification (CRV) [11,36,65,13] with the aim to foster the comparison and evaluation of software runtime verification tools. In 2016 and 2018, together with other partners of ARVI, we have also started to organize the two first of a series of Schools on RV [21,30].

2 A Taxonomy of Runtime Verification

Runtime Verification (RV) has grown into a diverse and active field during the last 15 years and has stimulated the development of numerous theoretical frameworks and tools. The paper in [34] presents a high-level taxonomy of RV concepts and use it to classify RV tools. The classification and discussion related to RV tools is beyond the scope of this report. We instead briefly recall the main points of the classification and refer to [34] for further details. We also do not discuss the application area part of the taxonomy, as applications of runtime verification is the object of study of another working group. The taxonomy provides a hierarchical organization of the six major concepts used in the field and serves to classify several of the existing tools. In this report, we report only on the first two levels for readability and brevity reasons:

*Specification.*A specification indicates the intended system behavior (property), that is *what one wants to check* on the system behavior. It is generally 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. A specification itself can be either implicit or explicit. An implicit specification is used in a runtime verification framework when there is a general understanding of the particular desired behavior. An explicit specification is one provided by the user of the runtime verification framework.

*Monitor.*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

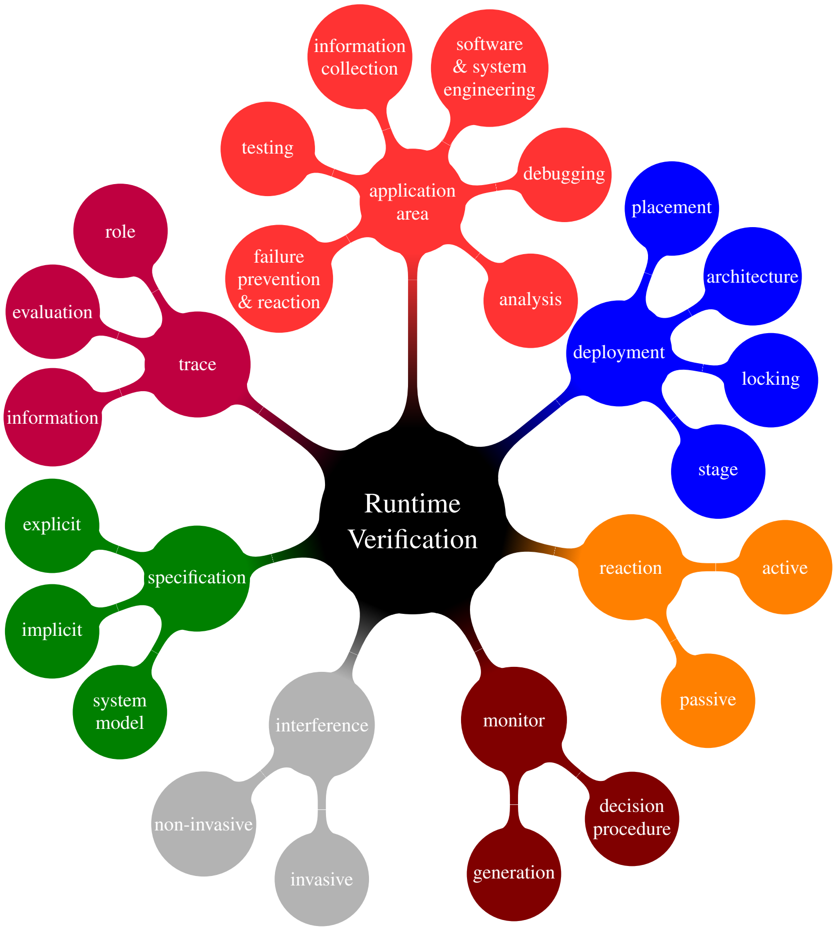

Fig. 1. Mindmap overviewing the taxonomy of Runtime Verification [34].

verification process. A monitor implements a semi-decision procedure which produces the expected output (either the related information for an implicit specification or the specification language output for an explicit specification). Note, the monitor may run forever without producing a verdict. Monitors must be generated from a specification.

*Deployment.*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. 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. The architecture of the monitor may be centralized (e.g., in one monolithic procedure) or decentralized (e.g., by utilising communicating monitors).

*Reaction.*By reaction, we refer to how the monitor affects the execution of the system. Reaction is said to be passive when the monitor does not influence or minimally in-

fluences the initial execution of the program. Reaction is said to be active when the monitor affects the execution of the monitored system.
*Trace.*The notion of trace appears in two places in a runtime verification framework and this distinction is captured by the role concept. A monitor can receive different sorts of information from a system (e.g., events, states, or signals). The system evaluation of the system may refer either to specific points in time or refer to intervals of time.
*Invasive.*In absolute, a non-invasive monitoring framework being impossible, the distinction between invasive vs non-invasive better corresponds in reality to a spectrum. There are two sources of interference for a monitoring framework with a system: the effect of the instrumentation applied to the system and the monitor deployment.

3 Instrumentation

The term instrumentation refers to the mechanism employed to probe and to extract signals, traces of events and other information of interest from a software or hardware system during its execution. Instrumentation determines what aspects of the system execution are made visible (to the monitor) for analysis, in terms of what elements of the computation is reported (e.g., computation steps and the data associate with them), and the relationships between the recorded events (e.g. provide a partial or total ordering, or guaranteeing that the reported event order corresponds to the order in which the respective computational step occurred). Instrumentation also dictates how the system and the monitor execute in relation to one another in a monitoring setup. It may either require the system to terminate executing before the monitor starts running (offline), interleave the respective executions of the system and the monitor within a common execution thread (online), or allocate the monitor and the system separate execution threads (inline vs outline); instrumentation may dictates how tightly coupled these executions need to be (synchronous vs asynchronous).

The choice of instrumentation techniques depends on the type of system to be monitored. For example, monitoring hardware system may require probing mixed-analog signals using physical wires, while for software the instrumentation method is strictly related to the programming language in which the software is implemented or to the low-level language in which it is compiled (i.e., bytecode, assembly, etc.).

The following instrumentation mechanisms exist:

Logging or manual instrumentation Most systems already log important actions. Typically, the level of detail of these logs is configurable. When a system is configured to log data a high level of detail, the log may contain enough information to derive a verification verdict from its data [49,58,42]. If this is not the case, additional data has to be obtained by manual instrumentation (manually inserting logging code). Furthermore, the format of the data is typically unstructured or semi-structured, and typically have to be pre-processed before they can be used for monitoring [72,69]. The advantages of this approach are that the data is typically already available, and no special tools have to be set up.

Code instrumentation (transpilation/weaving) With code instrumentation, the code of the program is modified such that the necessary statements to capture the information for runtime monitoring are inserted. This modification is usually automatic and supported by a tool. This has the advantage that it is possible to cover similar properties in a uniform way across a large program, as the code is modified in a systematic and automatic way. Furthermore, this technique is almost unlimited w. r. t. the amount and type of data it can access, as code instrumentation can be very fine-grained.

The tools that modify the code are often called transpilers (if the source code is modified) or weavers (if source or compiled code is modified). Transpilation occurs right before compilation, whereas compiled code can be modified after compilation, or at load time. In all of these cases, the interplay of various stages of building and deployment of software may not readily accept a code instrumentation stage without adaptation, due to actions such as code generation, dynamic loading of libraries, other code rewriting frameworks such as Spring [47], etc. While the result of source-to-source transformed can be readily inspected, weavers for compiled code are more flexible and also work without the source. However, it can be difficult to ensure that all information is captured fully and correctly, without altering the original behavior of the program.

Popular transpilers can be implemented as libraries [62], term rewriting systems [7,70], compiler extensions [52,38], domain-specific languages [50], or even transformation generators [51]. Weavers include domain-specific languages such as AspectJ [48] or AspectC++[68], but also libraries such as ASM [17]. Code instrumentation is also often provided by custom tools that have been designed with the given verification task in mind [66,59,67].

Call interception When analyzing activity such as input/output, which is accessed via existing libraries, the easiest way to observe these actions is by overloading the libraries with a wrapper. The wrapper then intercepts the call and updates the runtime monitor before or after calling the original library code. This approach differs from code instrumentation in that code is not modified throughout the system, but instead, the modification is made by intercepting calls systematically, and providing additional functionality before or after each function call.

Typical mechanisms to achieve this are the usage of the linker to replace a library call with a wrapper [18], kernel modifications [6], the boot-classpath in Java (up to Java 8) [60] or Java's module system [10]. This approach is less flexible than other approaches, in that it can only modify the behavior of code at function call boundaries, but it is easy to confine the modifications to small parts of the system. However, not all platforms have a straightforward mechanism of keeping the unmodified version and delegating a call to the original code inside the wrapper; for instance, older versions of Java completely replaced the overloaded library without any way of accessing the old code. Furthermore, the mechanisms to overload libraries are very specific to each platform, so tools using this technique are not portable.

Execution environment Many execution environments have interfaces with events from program execution can be obtained. These interfaces include the Java Virtual Machine Tool Interface (JVMTI [61]) and the LTTng interface for the Linux kernel [25]. Typically, the use of these mechanisms requires access to the data structures of the execution environment itself. Monitoring code is therefore not written at the level of the "guest" language (such as Java in the case of JVMTI), but at the level of the "host" language, the language in which the virtual machine is written in. This makes such monitoring approaches harder to use and less portable. Finally, execution events may even be generated by custom hardware.

Table 1. Advantages and disadvantages of different approaches

Approach	**Advantages and disadvantages**
Logging	+ Much data already available; no special tooling needed. − Data format is unstructured; available data may be limited; ad hoc.
Code instrumentation	+ Systematic, flexible approach. − Difficult to use and test; may interfere with original program.
Call interception	+ Limited modifications; good performance. − Not portable; limited to the interfaces of libraries.
Execution environment	+ Fast; no modification of the program needed. − Difficult; not portable; limited to given data.

This technique has the advantage that it works in an unmodified execution environment, and typically offers the least amount of overhead of all approaches. However, even though modern environments offer a large range of data through these tool interfaces, they are still restricted to a predefined set of data.

Therefore, the option to use a special or modified execution environment is sometimes used. A modified execution environment may change parts of the kernel [6] or use a specific virtual machine that generates a wider range of events [71], or a debugger that can inspect (and even modify) data at a finer level of detail than otherwise possible [46]. Unlike standard environments, performance and stability of runtime monitoring in these special environments depend on the modifications needed to obtain the data.

Finally, a non-standard execution environment may even involve special hardware access ports designed for monitoring [44]. In this case, there is often no overhead, and at the software layer, the execution environment is indistinguishable from a standard environment.

Table 1 gives an overview of the different types of approaches, and their pros and cons. In [14], we further explain these concepts in two dedicated sections for hardware and software instrumentation.

4 Interplay between Runtime Verification and other Verification Techniques

4.1 Static Techniques and Runtime Verification

As opposed to runtime verification, static verification techniques are used to analyse and prove properties of all possible executions of programs. Two prominent families of static verification techniques are model checking and deductive verification. In the following, we will discuss both of them, together with their respective relation to runtime verification.

Model Checking and Runtime Verification A popular verification technique besides runtime verification is *model checking* [19]. While runtime verification checks a single execution of the underlying system, model checking considers all possible executions of the underlying system. As such, both techniques share a common goal, i.e., the verification of an underlying system, but they have different features. In general, the two techniques may be combined. Let us briefly discuss the methodological combinations of the two techniques as well as their formal relationship.

At a first sight, model checking seems to provide stronger correctness guarantees than runtime verification as model checking considers all possible executions of a system. However, model checking still greatly suffers from the so-called state-space explosion problem limiting its application only to parts of or abstractions of an underlying system. As such, it makes perfectly sense to combine both verification techniques. In [56], several combinations are discussed, which we briefly summarize here.

- The verification result obtained by model checking is often referring to a model of the real system under analysis but not to the actual byte code, CPU etc.
- However, the implementation might behave slightly different than predicted by the model. Runtime verification may then be used to easily check the actual execution of the system, to make sure that the implementation really meets its correctness properties. Thus, runtime verification may act as a partner to model checking.
- Often, some information is available only at runtime or is conveniently checked at runtime. For example, whenever library code with no accompanying source code is part of the system to build, only a vague description of the behavior of the code might be available. In such cases, runtime verification is an alternative to model checking.
- The behavior of an application may depend heavily on the environment of the target system, for which certain assumptions have been taken at model checking time. Runtime verification allows to check such assumptions at runtime and may raise an alarm if one of the assumptions does not hold. Here, runtime verification completes the overall verification.
- In the case of systems where security is important or in the case of safety-critical systems, it is useful also to monitor behavior or properties that have been statically proved or tested, mainly to have a double check that everything goes well: Here again, runtime verification acts as a partner of model checking.

However, besides the methodological combination of runtime verification and model checking, there is a formal combination possible, as described in [53]. Let us recall the main idea here while we refer to the previously mentioned reference for formal details.

Especially in online runtime verification, an execution of the underlying system is checked against a correctness property letter by letter. To reuse for example linear-time temporal logic (LTL) defined over infinite traces [63], the verdict for the finite execution is obtained by considering all possible infinite continuations of the execution. If with all such extensions the LTL formula yield either only *true* or only *false*, the corresponding verdict is that for the finite execution seen so far. Otherwise, the verdict is *don't know?* to signal that the current excution does not allow a precise verdict anymore. This approach yields the so-called *anticipatory* semantics introduced in [15].

In [53], the idea was pursued that rather considering all possible infinite extensions of the current execution, only those are taken into account that yield runs of the overall system. Of course, a system model has to be at hand to understand which extension are possible at all. However, let us consider this idea for the empty execution, i.e., when the system has not even started: Then, we have to check whether all runs of the underlying system either satisfy or falsify the property at hand. The first question is actually the model-checking problem, and of course, if we have answered the model checking problem before we even start the execution of the system, the need for runtime verification vanishes. Now, assume that we consider an over-approximation of the underlying system, that is, a system having more runs/behavior, but smaller/less states. We still implicitly solve the model checking problem, indeed after every partial execution, yet for an abstract system. Thus, if the answer is that there is no violation possible any more (*true*), one can stop monitoring. If, however, the model checking answer is *false*, there might be a bug in the system, which may be found by further monitoring. Hence, by considering an over-approximation of the underlying system, we get best out of both worlds: we combine efficient runtime verification techniques with taking part of the system into account, hereby sharpening the verdict obtained by runtime verification. By tuning the abstraction, we can adjust whether the focus is on model checking or rather on runtime verification. In other words, by controlling the abstraction, we are able to slide between model checking and runtime verification. See [53] for further details.

Deductive Verification and Runtime Verification Deductive verification techniques are used to verify data-oriented, functional properties of code units (such as methods/procedures or classes), specified often in languages tailored to the programming language at hand (like, for instance, the Java Modeling Language —JML). Verification is typically done by reasoning directly about the source code, using a program logic like for instance Hoare logic or dynamic logic. Deductive program verification has been around for about 40 years, however, a number of developments during the last decade brought dramatic changes to how deductive verification is being perceived and used. Among the state of the art efforts is the KeY tool for Java source code verification [2]. Deductive Verification has been extensively used to verify properties focusing on the system's data at specific points of the execution (like method entries and exits). Runtime verification, on the other hand, has been extensively used to verify trace properties with reasonable overheads (e.g., automata based or temporal). As both approaches work on the concrete system level, without abstraction, there is great potential for combining them. Ideally, (sub)properties which are a bottleneck for static verification shall be addressed by runtime verification, whereas properties which require high overhead for runtime checking shall be addressed by static verification. For that, however, we need specification languages allowing the expression of combined data- and control-flow properties in such a manner that they can be effectively decomposed for the application of different verification techniques. This has been exemplified in the StaRVOOrS approach [3], which provides a specification language combining data- and control-oriented aspects, and combines a deductive and a runtime verification tool to optimise runtime verification by (partial) results from (static) deductive verification.

4.2 Model Learning and Runtime Verification

Many verification techniques rely on the presence of a formal specification or a model of the system under verification. In many cases, such a model is not available. Model learning is an approach to automatically infer the model, by observing traces of the system. This can either be done passively, by purely observing the system, or actively, by steering the system execution into interesting areas.

Bertolino et al. [16] have suggested a combination of active learning and monitoring, where the produced system traces are continuously checked for conformance with the currently learned model. In case of non-conformance, the model will be updated to reflect the newly observed behaviour. Isberner et al. [45] proposed the TTT algorithm, which attempts to reduce the length of the observed counter example, to optimize the learning from long traces, as observed by runtime verification.

Both these approaches consider a continuous learning approach, where the learned model can be updated at any point of the execution. Thus, if a bug is observed, it will be incorporated into the model. Contrary to this, one could propose a two-step approach combining model learning and monitoring: in the first step the model is learned, using any of the existing algorithms. In the second step, this learnt model is considered to be correct and used as a basis for the monitoring. If, at any point during the runtime execution, an incorrect behaviour is encountered, it is reported as an anomalous behavior, rather than incorporated into the model. The main goal of such an approach would be the detection of transient errors, i.e., errors that only occur sporadically or errors which only occur under certain conditions in the environment.

Several techniques investigated how to learn different kinds of models that can enable the analysis of various classes of behaviors. For instance, Mariani et al. investigated how to learn finite state models and likely program invariants to analyze executions in component-based systems [57]. Similarly, Pradel and Gross used learnt finite state models to analyze API misuses [64]. Recently, Grant et al. investigated how to infer and check assertions in distributed systems [39].

4.3 Testing and Runtime Verification

To date, testing is by far the most commonly used technique to check software correctness. In essence, testing attempts to generate a number of test cases and checks that the outcome of each test case is as expected. While this technique is highly effective in uncovering bugs, it cannot guarantee that the tested system will behave correctly under all circumstances.

Typically, testing is only employed during software development; meaning that software has no safety nets during runtime. Conversely, runtime verification techniques are typically used during runtime to provide extra correctness checks but little effort has been done to integrate it with the software development life cycle. For this reason little or no use of it is made in contemporary industry.

At a closer look, the two techniques — testing and runtime verification — are intimately linked: runtime verification enables the checking of a system's behaviour during runtime by listening to system events, while testing is concerned with generating an adequate number of test cases whose behaviour is verified against an oracle. Thus, in summary one could loosely define runtime verification as event elicitation behaviour checking, and testing to be test case creation behaviour checking.

Through this brief introduction of testing and runtime verification one can quickly note that behaviour checking is common to both. Given this overlap between two verification techniques, one is surprised to find that in the literature the two have not been well studied in each other's context, even though there were a few isolated attempts where RV and testing principles were jointly used [4,5,31,32].

The paper [20], outlines three ways in which this can be done: (i) one where testing can be used to support runtime verification in the creation of more reliable tools, (ii) another where the two techniques can be used together in a single tool such that the same set of properties can be tested and runtime verified, and (iii) a third approach where

Table 2. Comparison of Verification Techniques (simplified)

Runtime Verification	**Static Verification**	**Properties**	**Specifications**
Runtime Trace Checking	Model Checking	valid traces (+ some data)	temporal logics, automata, regular languages (+ extensions)
Runtime Assertion Checking	Deductive Verification	valid data in specific code locations (+ some trace info)	first-order assertion languages (+ extensions)

runtime verification can be used to support testing in gathering information regarding areas of the code which are observed executing at runtime.

4.4 Runtime Assertion Checking and Runtime Verification

Another area strongly connected to runtime verification (RV) is *runtime assertion checking* (RAC). Strictly speaking, RAC is a special case of RV. Nonetheless, the characteristics are so different from most approaches to RV that RAC is sometimes seen as a field of its own. In any case, we here take the approach which classifies RAC as an RV technique, whereas the other, mainstream approaches to RV are identified as *runtime trace checking* (RTC). We give an overview of this classification in Fig. 2, also relating it to the major areas of static verification, model checking and deductive verification. Note that we are deliberately simplifying the presentation. Doing full justice to all verification approaches is not in scope of this section. Rather we want to exhibit some basic characteristics.

RV mostly focuses on properties of execution *traces* of some system, so most of RV fall under runtime trace checking (RTC). These properties may be specified using different formalisms, of which temporal logics, automata, and regular languages are prominent examples. In what concerns properties and specification approaches, RTC has therefore similarities with Model Checking. Most of this entire document is about RTC.

On the other hand, properties which are typically addressed by runtime assertion checking (RAC) focus on conditions on the *data*, in specific code locations. RAC comes with assertion languages, to formulate assertions to be 'placed' at source code locations. These assertions often use concepts from first-order logic to constrain which data is valid in the respective code locations. RAC is therefore richer than RTC when it comes to properties of the data, but less expressive when it comes trace related properties.

A prominent example of a runtime assertion checker is OpenJML[13], which supports the checking of JML [43] assertions while executing (instrumented) Java applications. Other examples of RAC tools are SPARK [8], and SPEC# [9], supporting variants of Ada and C#, respectively.

One has to say that RAC techniques, because of the expressiveness of the used specification languages, are often too slow to be used for post-deployment RV, and are

[13] www.openjml.org

therefore better suited for the debugging phase. This may be another reason why RAC is not always counted as an RV technique. However, recent developments show that RAC can be very much optimized by combining static and runtime verification [3].

5 Challenges in Monitoring Quantitative and Statistical Data, beyond Property Violation

Since runtime verification has traditionally borrowed techniques, and in particular specification languages, from static verification the specifications from which monitors are extracted are typically temporal logics and similar formalisms. In turn, the outcome of a monitoring process is typically a Boolean verdict (or sometimes an enriched Boolean verdict to convey non-definite answers). At the same time, decision problems on specifications (like equivalence vacuity, entailment, etc.) are decidable as these problems have been studied in static verification. However, if one is willing to sacrifice the decidability of the these decision procedures, there is the possibility of designing "richer" logics. One can perform runtime verification activities as long as there is a formal procedure to generate monitors from specifications, and an associated evaluation method for evaluating the generated monitors against an input trace. Taking this abstract viewpoint on monitoring allows to consider languages that process rich data and generate richer outcomes.

Consequently, we suggest here two directions in which runtime verification can be extended in terms of how rich the data that the monitors handle is. The first direction is about the data the monitor processes and manipulates. The second direction is about richer outcomes from the monitoring process.

5.1 Richer Data for Monitors

Two research areas related to runtime verification that study how to handle sequences of events with rich data, and that can produce rich data as outcome, are Complex Event Processing (CEP) and Data Stream Management Systems (DSMS); see [23] for a modern survey. Complex Event Processing considers distributed system that processes flows of events from different sources, finding correlations and producing derived events as a result. The main difference between CEP and RV (even the vision of RV with rich data that we advocate in this section) is that CEP does not provide a formal specification language with formal semantics, but instead an infrastructure to evaluate event processors. Similarly, DSMS borrow many techniques from Data Bases with the main difference that queries are supposed to process the flow of events without storing all events (even though many DSMS do create a storage with a time index). Typically, DSMS process aggregations over flows of input events where the main concerns are efficiency in the evaluation process and statistical properties of the output (instead of logical correctness). One can potentially map the formal specifications that we use to generate monitors in RV to queries (for DSMS) and to processors (in CEP), these areas do not provide formal translations nor a formal semantics of the execution platforms with guarantees on the order of arrival of events.

The book chapter "Monitoring Events that Carry Data" [41] [12] reviews the research landscape on runtime verification techniques for traces that contain rich data, but we envision that richer and richer extensions will be investigated in years to come.

5.2 Richer Outcomes from the Monitoring Process

Another important aspect that has not been extensively studied in runtime verification is richer verdicts. If one considers monitors as transformers from traces into useful "summaries" of information about the trace, there are many possibilities. For example, in the area of signal temporal logics, there is the notion of "robustness", which is a quantity that measures "how much" an observed trace matches a given specification (in specialized logics for the domain of signal processing like STL, MTL, etc.). A value of 0 can indicate that a full violation is detected, while values close to 0 (like 0.4) can indicate that a violation is close to occur. This richer outcome allows to determine hot robustly a trace is satisfied or violated. Potentially, the outcome of a monitor could be a pruned trace that allows further analysis to be performed, for example to determine the root of the error. Finally, there are few works that consider the possibility of processing (statistically) incorrect or missing data. One challenge for the near future is to explore different rich outcomes that the monitors can generate and the trade-offs involved in the evaluation of these monitors.

One potential direction to attack this challenge is Stream Runtime Verification (SRV), which was conceived to trigger richer verdicts beyond YES/NO answers. SRV, pioneered by the tool Lola [24], proposes to use streams to separate two concerns: the algorithms to perform evaluations of temporal properties on input traces, and the data collected during the evaluation. The key insight is that many algorithms, for example to check LTL properties (and similar formalisms proposed to express monitors in the RV community) can be easily generalized to compute richer verdicts. As a simple example, one can use the same algorithm that searches for a violating position in the past to compute the number of offending positions. Similarly, one can compute the longest response time to a request, or the average response time (and not just the fact that all requests are answered). Modern extensions of stream runtime verification enrich the basic setting to parameterized properties [37] so the input stream can be classified according to different objects. Applications include network monitoring [37], security assurance of unmanned aerial vehicles [1] and real-time SRV for the dynamic analysis of timed-event streams from low-level concurrent software [54]. All these results were motivated by challenging domains for which the ability of SRV to handle quantitative data and produce rich verdicts proved to be valuable.

Another potential direction to attack this challenge consists in using *decentralised specifications* [26], pioneered by the tool THEMIS [27]. While simple specifications can be expressed with traditional specification formalisms such as LTL and automata, accounting for hierarchies quickly becomes a problem. Informally, a decentralized specification considers the system as a set of components, defines a set of monitors, additional atomic propositions that represent references to (the verdict of other) monitors, and attaches each monitor to a component. Each monitor is a Moore automaton where the transition label is restricted to only atomic propositions related to the component on which the monitor is attached, and references to other monitors. Decentralized specifications were successfully applied to the monitoring of smart homes [28] where they proved to allow a better scaling than with traditional specification formalisms.

References

1. Adolf, F., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on UAS. In: Lahiri, S.K., Reger, G. (eds.) Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10548, pp. 33–49. Springer (2017)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
3. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. Formal Methods in System Design 51(1), 200–265 (2017)
4. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M.R., Pasareanu, C.S., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theor. Comput. Sci. 336(2-3), 209–234 (2005)
5. Artho, C., Biere, A., Seidl, M.: Model-based testing for verification back-ends. In: Veanes, M., Viganò, L. (eds.) Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7942, pp. 39–55. Springer (2013)
6. Artho, C., Suzaki, K., Hagiya, M., Leungwattanakit, W., Potter, R., Platon, E., Tanabe, Y., Weitl, F., Yamamoto, M.: Using checkpointing and virtualization for fault injection. International Journal of Networking and Computing 5(2), 347–372 (2015)
7. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on Java. In: International Conference on Rewriting Techniques and Applications. pp. 36–47. Springer (2007)
8. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
9. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS'05. Lecture Notes in Compute Science, vol. 3362, pp. 49–69. Springer (2005)
10. Bartlett, N., Beaton, W., Dockter, H., Ellison, T., Forax, R., Lee, B., Lloyd, D., Reinhold, M., Scholt, R.: Java platform module system JSR (376). http://openjdk.java.net/projects/jigsaw/spec/ (2017), accessed: 2018-11-20
11. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Proc. of RV 2014: the 5th International Conference on Runtime Verification. LNCS, vol. 8734, pp. 1–9. Springer (2014)
12. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018)
13. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. International Journal on Software Tools for Technology Transfer (Apr 2017)
14. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci and Falcone [12], pp. 1–33
15. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14:1–14:64 (2011)
16. Bertolino, A., Calabrò, A., Merten, M., Steffen, B.: Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. ERCIM News 2012(88) (2012)
17. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems. Grenoble, France (2002)

18. Chamberlain, S.: Using LD, the GNU linker. https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html (1994), accessed: 2018-11-20
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
20. Colombo, C.: Combining testing and runtime verification (computer science annual workshop). Tech. Rep. CS2012-03, Department of Computer Science, University of Malta (2012), available from http://www.um.edu.mt/ict/cs/research/technical_reports
21. Colombo, C., Falcone, Y.: First international summer school on runtime verification - as part of the arvi COST action 1402. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 17–20. Springer (2016)
22. Colombo, C., Leucker, M. (eds.): Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, Lecture Notes in Computer Science, vol. 11237. Springer (2018)
23. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44(3), 15:1–15:62 (2012)
24. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning. pp. 166–174. IEEE Computer Society (2005)
25. Desnoyers, M., Dagenais, M.R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: OLS (Ottawa Linux Symposium). vol. 2006, pp. 209–224 (2006)
26. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. pp. 125–135. ACM (2017)
27. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. pp. 372–375. ACM (2017)
28. El-Hokayem, A., Falcone, Y.: Bringing runtime verification home. In: Colombo and Leucker [22], pp. 222–240
29. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)
30. Falcone, Y.: Second school on runtime verification, as part of the arvi COST action 1402 - overview and reflections. In: Colombo and Leucker [22], pp. 27–32
31. Falcone, Y., Fernandez, J., Mounier, L., Richier, J.: A test calculus framework applied to network security policies. In: Havelund, K., Núñez, M., Rosu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4262, pp. 55–69. Springer (2006)
32. Falcone, Y., Fernandez, J., Mounier, L., Richier, J.: A compositional testing framework driven by partial specifications. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4581, pp. 107–122. Springer (2007)

33. 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 Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
34. Falcone, Y., Krstic, S., Reger, G., Tratel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) Proceedings of the 18th International Conference on Runtime Verification (2018), submitted
35. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 103–134. Springer (2018)
36. Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification CRV 2015. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9333, pp. 405–422. Springer (2015)
37. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 152–168. Springer (2016)
38. Free Software Foundation, Inc.: Using the GNU compiler collection: Instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (2018), accessed: 2018-11-20
39. Grant, S., Cech, H., Beschastnikh, I.: Inferring and asserting distributed system invariants. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1149–1159. ACM (2018)
40. Havelund, K., Goldberg, A.: Verify your runs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005)
41. Havelund, K., Reger, G., Thoma, D., Zalinescu, E.: Monitoring events that carry data. In: Bartocci and Falcone [12], pp. 61–102
42. Hu, C., Neamtiu, I.: Automating gui testing for android applications. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 77–83. ACM (2011)
43. Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java Modeling Language. In: Deductive Software Verification—The KeY Book, Lecture Notes in Compute Science, vol. 10001. Springer (2016)
44. IEEE: Designing for on-board programming using the IEEE 1149.1 (JTAG) access port (2008)
45. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014)
46. Jakse, R., Falcone, Y., Méhaut, J.F., Pouget, K.: Interactive runtime verification—when interactive debugging meets runtime verification. In: ISSRE17-28th International Symposium on Software Reliability Engineering (2017)
47. Johnson, R., Hoeller, J., Donald, K., Sampaleanu, C., Harrop, R., Risberg, T., Arendsen, A., Davison, D., Kopylenko, D., Pollack, M., et al.: The Spring framework–reference documentation. Interface 21, 27 (2004)

48. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: European Conference on Object-Oriented Programming. pp. 327–354. Springer (2001)
49. Kingsbury, K.: Jepsen: ZooKeeper. https://aphyr.com/posts/291-jepsen-zookeeper (2013), accessed: 2018-10-19
50. Klint, P., Van Der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 168–177. IEEE (2009)
51. Kuipers, T., Visser, J.: Object-oriented tree traversal with JJForester. Electronic Notes in Theoretical Computer Science 44(2), 1–25 (2001)
52. Lee, S.I., Johnson, T.A., Eigenmann, R.: Cetus–an extensible compiler infrastructure for source-to-source transformation. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 539–553. Springer (2003)
53. Leucker, M.: Sliding between model checking and runtime verification. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7687, pp. 82–87. Springer (2012)
54. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. pp. 1925–1933. ACM (2018)
55. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
56. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
57. Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. IEEE Trans. Software Eng. 37(4), 486–508 (2011)
58. Naldurg, P., Sen, K., Thati, P.: A temporal logic based framework for intrusion detection. In: International Conference on Formal Techniques for Networked and Distributed Systems. pp. 359–376. Springer (2004)
59. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan notices. vol. 42, pp. 89–100. ACM (2007)
60. Oracle: How classes are found. https://docs.oracle.com/javase/8/docs/technotes/tools/findingclasses.html (2018), accessed: 2018-11-20
61. Oracle: JVM tool interface 11.0.0. https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html (2018), accessed: 2018-11-20
62. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A library for implementing analyses and transformations of Java source code. Software: Practice and Experience 46, 1155–1179 (2015)
63. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977), https://doi.org/10.1109/SFCS.1977.32
64. Pradel, M., Gross, T.R.: Leveraging test generation and specification mining for automated bug detection without false positives. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. pp. 288–298. IEEE Computer Society (2012)
65. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification - CRV 2016. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 21–37. Springer (2016)

66. Rubanov, V.V., Shatokhin, E.A.: Runtime verification of linux kernel modules based on call interception. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on. pp. 180–189. IEEE (2011)
67. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multi-threaded programs. SIGOPS Oper. Syst. Rev. 31(5), 27–37 (1997)
68. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: an aspect-oriented extension to the C++ programming language. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. pp. 53–60. Australian Computer Society, Inc. (2002)
69. Tang, L., Li, T., Perng, C.S.: LogSig: Generating system events from raw textual logs. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 785–794. ACM (2011)
70. Visser, E.: Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5. In: International Conference on Rewriting Techniques and Applications. pp. 357–361. Springer (2001)
71. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated software engineering 10(2), 203–232 (2003)
72. Zhu, K.Q., Fisher, K., Walker, D.: Incremental learning of system log formats. ACM SIGOPS Operating Systems Review 44(1), 85–90 (2010)