# Introduction to Runtime Verification

Ezio Bartocci, Yliès Falcone, Adrian Francalanza, Giles Reger

# Introduction to Runtime Verification

Ezio Bartocci[1], Yliès Falcone[2], Adrian Francalanza[3], and Giles Reger[4]

[1] TU Wien, Austria
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP,
Laboratoire d'Informatique de Grenoble, F-38000 Grenoble, France
[3] University of Malta, Msida MSD2080, Malta
[4] University of Manchester, Manchester, UK

**Abstract.** The aim of this chapter is to act as a primer for those wanting to learn about Runtime Verification (RV). We start by providing an overview of the main specification languages used for RV. We then introduce the standard terminology necessary to describe the monitoring problem, covering the pragmatic issues of monitoring and instrumentation, and discussing extensively the monitorability problem.

## 1 Introduction

The field of Runtime Verification (RV) has been, and is still, referred to by many names such as runtime monitoring, trace analysis, dynamic analysis etc. 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. At this early point in this chapter we would like to note that the community is not in agreement about the various meanings of certain terminology, such as the difference between *runtime verification* and *runtime monitoring*. We take a popular interpretation in this chapter, but the reader will most likely encounter alternative views in the literature.

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 price 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. This is also the main activity considered in this chapter. 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 handbook. 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 Runtime Verification (CRV) [17, 21] with the aim to foster the comparison and evaluation of software runtime verification tools. In the same year, a European scientific network for the COoperation in Science and Technology (COST) on *Runtime Verification beyond Monitoring (ARVI)* was approved and funded within the European framework programme Horizon 2020. ARVI currently includes the participation of scientists from 26 European countries and Australia. In 2016, together with other partners of ARVI, we have also started to organize the first of a series of Schools on RV. However, it is worth noting that the research on monitoring techniques has been around for a very long time and is present in other communities where it is not referred to in the same terms as it is here, even if the process is the same.

In this chapter we introduce the field of RV covering the basic concepts and the standard notions of monitoring. We have not attempted to make a full survey of all related work but we refer to the main relevant literature [76, 98, 110, 137]. When considering how to check whether the runtime behaviour of a system conforms to some specification there are three necessary steps to be taken:

1. *Specifying (Un)Desired System Behaviour.* Section 2 considers how system behaviour can be abstracted in terms of events and traces and how specification languages can be used to describe properties of such traces.
2. *Producing a Monitor from a Specification.* Section 3 considers the monitoring problem and various issues that must be dealt with during monitoring.
3. *Connecting a Monitor to a System.* Section 4 considers how various *instrumentation* approaches can be used to extract the information necessary for monitoring from a running system.

We are also interested in the question of *what can and cannot be monitored*; this is addressed in Section 5. Even though this question seems more theoretical, it determines what sorts of properties can be handled with runtime verification. We provide an overview of all the chapters of this handbook in Section 6 and we conclude in Section 7.

## 2 Formal Specification of the System Behaviour

This section introduces the reader to different formal approaches to describe the expected behaviour of a system. We start by presenting various abstractions enabling to reason about the system behaviour at different level of detail. We then present some specification languages after having discussed first some general properties of these formalisms.

*Example 1 (Traffic Lights).* Throughout this section we choose as our running example a traffic light system. This system consists of three lights of different colors: *green*, *red*, *yellow*. We then consider how to specify the expected behaviour of such a system using different formalisms.

### 2.1 The Abstract Notion of Behaviour

When we consider the behaviour of a system we are referring to the way the system changes over time, by updating some internal state, taking some internal or external action, or affecting the environment in some way. We typically describe this behaviour in terms of the observations we can make about it. There are two general kinds of observations we can make: either we inspect some snapshot of the current *state* of the system at a particular time, or we record certain *actions* or *state changes* made by the system (where the system in question may include the environment). Below we describe how we will abstract systems in terms of the observations (events) we can make about them over time (giving a trace) and how we describe behaviour using these abstractions.

*Events.* We will call any kind of observation about the system an *event*. In the simple case an event is a name for something that can happen, for example lightTurnsGreen, lightBrightnessIs80 or pedestrianButtonPressed. In the more complex case an event could be structured, containing data values of interest. We do not cover this complex case here but it is discussed in Chapter 3. Note that we make a distinction between events as syntactic elements and what they denote semantically. For example, an event temperatureLimit may correspond to a sensor detecting that the temperature is at or above $20°C$. We separate the representation of this event and the process of recording/detecting and reporting it (Section 4 describes how we practically observe systems).

In this presentation events are discrete atomic entities, but there are two alternative views that are taken in various work. Firstly, some work considers an event as having a *duration*, i.e., a start and end time. This can be easily translated into the setting where events are atomic by introducing associated start and end events. For example, we might have the event *lightGreen* at one level of abstraction, but *lightGreenOn* and *lightGreenOff* at another level of abstraction. Secondly, in languages specified over *continuous time* events can be viewed as *signals* that can be queried for a value. We discuss this setting in more detail later. Where an alternative interpretation of event is being taken we will be explicit in the text.

We will call a system's observable events of interest its *alphabet*. We stress *of interest* as there are potentially an infinite number of different ways of describing an event but typically we are only interested in a small (at least usually finite) set of such events.

It is clear that the choice of events is fundamental in how much information we have about a system and the properties we can describe. Events can be defined at different levels of abstraction, be about internal or external behaviour, may cover the whole system or one specific part, and may not correspond directly with existing components or actions defined in the system. The choice of events is part of the specification process and will depend on what the end goal is.

We note that this choice of abstracting systems in terms of *events* rather than *states* is a distinction that is different from (but compatible with) the work of model checking. As with many things, not all work in RV uses the event abstraction and some work may view an execution as a sequence of observed states.

*Traces.* We use events to abstract a particular observation about a system. We abstract the behaviour of a single run of a system as a *trace*, which is a (finite) sequence of events (or sets of events) [126]. Clearly, an observable trace of a system must be *finite*, but it is sometimes useful to think about the possible *infinite* behaviours of a system. As discussed later, when viewing a trace as a finite prefix of some possible infinite behaviour we can ask whether the finite prefix can be extended to some acceptable infinite trace. Another key choice in structuring a trace is whether each point in the trace consists of a single event or a set of events. The single event view is often more straightforward, but it does not easily allow for settings where multiple observations may occur concurrently (and it does not make sense to coalesce them) or the exact ordering of events within a particular time frame is unclear (or unhelpful to enforce). This choice may seem arbitrar.y but it has an impact on the interpretation of specifications as those languages assuming a single event at a time implicitly include extra axioms, i.e., seeing one event precludes seeing any other. Finally, different approaches treat the notion of *time* differently. The order of events in a trace gives a qualitative notion of time, but it does not immediately capture any quantitative distance (in time) between events. We discuss the different approaches for embedding time into traces later.

*Properties and Specifications.* A *property* of a system can be abstractly described as a (possibly infinite) set of traces. A *specification* is a concrete (textual) object describing a property and therefore it denotes a set of traces. We have chosen to distinguish properties from specifications as the distinction can be important. However, this distinction is not universal in the literature. Note that the full behaviour, or intended behaviour, of a system may be given as a property. However, this will always be restricted by the event abstractions chosen and it is usually not the aim to describe total behaviour but key behaviour at a level of abstraction that is useful.

Using this separation we can have many specifications for a single property, but a property is unique and independent of a specification language. If the specification language is ambiguous (e.g., English) then the specific property being described may not be clear. Dealing with such ambiguities is a common issue in the specification process. Generally we expect a *specification language* to be unambiguous, at least in terms of the traces its specifications denote. We note that most work in the area conflate the notions of property and specification, and we may do so here. This is due to the fact the specification is the only object that exists concretely and it is often used to represent its underlying property.

A somewhat alternative distinction that some make between property and specification is that a property describes a unit of behaviour whilst a specification may capture many properties.

Much of the activity of RV considers *explicit* properties captured in some specification language. However, there are also many *implicit* properties covered by the field. A notable example of an implicit property is *deadlock avoidance* (see Chapter 2). When monitoring this property the property itself is not written in a specification language; instead specific ad-hoc algorithms are written to detect a violation of the property. Other examples of implicit properties are memory safety and bounds checking.

## 2.2 General Specification Language Features

In the following we discuss general features of specification languages used for runtime verification. We do not aim to present a taxonomy of languages, but instead aim to introduce some general concepts and terminology that is helpful when discussing such languages. See [99] for a more in-depth discussion of such features.

*Executable versus Declarative.* In some specification languages (e.g., state machines) the specification is directly executable whereas in other languages (e.g., temporal logic) it is more common to generate an executable object (monitor) from the specification. Languages where specifications are executable tend to have more straightforward monitoring algorithms. However, executable specifications also tend to be more low-level (operational) and less able to capture properties at a high level of abstraction. For example, it is usually more straightforward to combine specifications written declaratively, e.g., if the temporal logic formula $\varphi_1$ represents the normal behaviour of a traffic light system and formula $\varphi_2$ represents some special behaviour to be seen if a special emergency event occurs then the total behaviour should be $\varphi_1 \vee (\mathsf{emergency} \rightarrow \varphi_2)$. With automata this would either require monitoring mechanisms to allow the monitoring of multiple automata, or a construction on the automata leading to a complex automata that is difficult to read.

*Prefix Closure.* Consider the property that the yellow light is never on immediately after the red light. We might try to specify this using the regular expression

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

but under the standard semantics this does not accept the trace green yellow red. Our intention is for all *prefixes* of the describe language to be accepted. Such properties are *safety* properties and are common in system specification. Some specification languages assume prefix-closure, although most do not.

*When Language $\neq$ Property.* Typically, a concrete specification denotes a set of traces. Sometimes, for usability reasons, it might be useful for this language to not directly describe the property being specified, but be implicitly related to it. Therefore, this is less a feature of a language and more a feature of its usage. We give two examples here:

1. *Polarity.* A specification may capture *good* (desired, positive) behaviour or *bad* (undesired, negative) behaviour. Consider again the above property that the yellow light is never on immediately after the red light. Any trace *not* satisfying this property would match the regular expression

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

   which is, arguably, easier to read. When good behaviour is described a match represents *validation* of the desired property, whereas matching a specification describing bad behaviour represents a *violation* of that property.
2. *Suffix Matching.* Consider again the above property, the expression $(\mathsf{red} \mid \mathsf{green} \mid \mathsf{yellow})^*$ represents all possible traces and is, in some sense, redundant. In the interests of readability it would be more concise to simply write the expression

   $$\mathsf{red}\ \mathsf{yellow}$$

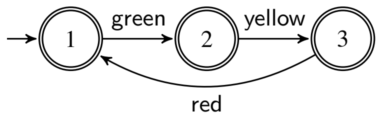
   and let it be understood that matching this expression against the *suffix* of a trace *violates* the desired property.

*Finite versus Infinite.* Some specification languages are more suited to specifying sets of finite traces (e.g., state machines) whereas other are more suited to specifying sets of infinite traces (e.g., temporal logic). As observations at runtime are necessarily finite this often leads to a mapping from a semantics over infinite traces to one over finite traces.

*Time.* As mentioned above, a totally ordered trace gives a qualitative notion of time, but not a quantitative one. Specification languages whose specifications denote such traces also only capture this qualitative notion of time. Notice that this qualitative notion is fragile as properties making use of this make assumptions about the level of abstraction events will be recorded at. Consider the property that the green light should be followed by the red light. Unless we are careful, the following two traces would not satisfy this property:

$$\tau_1 = \mathsf{green}\ \mathsf{green}\ \mathsf{red} \qquad \tau_2 = \mathsf{green}\ \mathsf{pedestrianButtonPressed}\ \mathsf{red}$$

Furthermore, if we want to check that the green light is on for 30 seconds it would be necessary to sample this light at a particular frequency and count the number of events seen. As discussed below, there are various methods for integrating quantitative notions of time into a specification language. Most commonly this is via explicit *clocks* in executable languages, or explicit *intervals* in declarative ones. With this quantitative notion one can now say how long the green light should be on and how soon the red light should come on once it goes off.



$$(\mathsf{green\ yellow\ red})^*(\mathsf{green} \mid \mathsf{green\ yellow} \mid \varepsilon)$$

$$\Box(\mathsf{green} \rightarrow \circ\mathsf{yellow} \wedge \mathsf{yellow} \rightarrow \circ\mathsf{red} \wedge \mathsf{red} \rightarrow \circ\mathsf{green})$$

**Fig. 1.** Illustrating the traffic light sequence property using a state machine, regular expression, and linear temporal logic formula.

*Data and Quantification.* A specification language may view events as atomic symbols or as structures containing *data*. Languages that consider data may do so in various ways, but this tends to be dependent on the underlying formalism. Temporal logics may be extended by standard notions of quantification [69, 53, 23], quantification over the *active domain* [26, 95, 141], or (more recently) with freeze quantifiers [19, 18, 63, 24]. An alternative approach is the use of *parametric trace slicing* [52, 127] to add a form of data quantification to a range of otherwise propositional formalisms (e.g., regular expressions and state machines). Specification languages making use of state machines often include the idea of transitions being labelled with guards and assignments [59, 12]. Finally, some formalisms, such as stream languages [81] and rule systems [16], have data manipulating features as standard. Chapter 3 considers such languages in more detail.

## 2.3 Specific Specification Languages

This section focusses on particular (families of) specification languages used for runtime verification. Figure 1 formalises a typical light sequence property of traffic lights in various basic languages described in this section.

**Temporal Logic** The most common family of specification languages used for runtime verification is temporal logic with the most basic and usual variant being *linear temporal logic* (LTL) [124].

*Linear Temporal Logic (LTL).* Future-time LTL introduces two basic modal operators: *Next* is written $\circ\varphi$ and means that $\varphi$ is true at the next point of the trace; and *Until* is written $\varphi_1 \,\mathcal{U}\, \varphi_2$ and means that $\varphi_1$ is true from the current point of the trace until $\varphi_2$ is true. These operators are used to define two (often more frequently used) operators: *Always* is defined as $\Box\varphi \equiv \varphi \,\mathcal{U}\, \mathit{false}$ and means that $\varphi$ should be true on every step of the trace from the current one onwards; and *Eventually* is defined as $\Diamond\varphi \equiv \neg\Box\neg\varphi$ and means that $\varphi$ is true at some point in the trace from the current point onwards. Some variations of LTL also introduce a notion of *Weak Until* that does not require $\varphi_2$ to eventually hold, only that $\varphi_1$ holds until it does (i.e., this may be infinitely often).

Past-time LTL has symmetric operators looking into the past e.g., *Previous* ($\bullet$) as the dual of *Next* and *Since* ($\mathcal{S}$) as the dual of *Until*. However, things are not quite this simple due to the finite nature of the past. It is typical to introduce a notion of *Weak Previous* ($\hat{\bullet}$) that is always true at the first state; it is then possible to define $\bullet\varphi = \neg\hat{\bullet}\neg\varphi$. It is common to consider a setting where both future-time and past-time operators are available.

In the runtime verification setting it is typical to consider *finite* traces only. As LTL has an infinite trace semantics it is necessary to provide an alternative finite trace semantics to deal with the end of trace. There are two main approaches to this:

- Providing an alternative semantics ensuring that $\Box\varphi$ is true at the end of a trace and $\Diamond\varphi$ is false. One way of achieving this is to add the dual of *Weak Previous* i.e., *Weak Next* and set up the semantics to preserve the identify $\Box\varphi = \varphi \wedge \hat{\circ}\varphi$. However, it is more common to define an alternative semantics directly without introducing $\hat{\circ}$ (see, for example, the early work in [128]).
- The finite trace is a finite prefix of some infinite trace. The truth of a formula on this finite prefix is defined by the possible *extensions* of that prefix i.e., it is true if all extensions make it true. This necessitates a multi-valued verdict domain. This idea is captured in $LTL_3$ [27] where a third verdict ? is given where some extensions are failing and some succeeding. This is called *impartiality* and means that some formulas (e.g., $\Box a$) can never be satisfied (there are always bad extensions) and dually some can never be violated (e.g., $\Diamond a$). These ideas relate to the notion of *monitorability*, discussed later.

There is an additional dimension that crosscuts both approaches, that of *anticipation*. The general idea is that if every extension of a finite trace leads to a single verdict then this verdict should be given to that finite trace. In the case where a purely finite trace semantics is given, the notion of anticipation is often captured by splitting the verdict domain into two forms of verdicts: strong verdicts reflecting an anticipatory result (all extensions) and weak verdicts reflecting the verdict to be given if the trace were to finish at the current point [16, 12].

*Interval Temporal Logic.* In LTL formulas are given over states or events i.e., distinct points in time. An alternative view, taken by interval temporal logic [48, 144], is to reason over *intervals*, i.e. pairs of points in time. Formulas in this logic may then use binary relations comparing intervals e.g. whether their start/end points are ordered, or whether one interval overlaps with, or is contained within, another. This presentation is generally not strictly more expressive than LTL as translations have been given into LTL [129].

*Variants with Time.* Standard temporal logics take a *qualitative* view of time i.e. they place an ordering on events but do not relate those events to the quantitative time line they occur within. There exist variants of LTL that add a notion of quantitative time via the extension of the underlying model (trace) with timestamps and an extension of the language constructs. Two notable variants are metric temporal logic [142] and timed LTL [29] which both use the notion of *intervals* to talk about ranges of time points. In MTL, temporal operators are annotated with discrete time intervals, e.g. $\varphi\ \mathcal{U}_{[3,7]}\ \psi$ states that $\psi$ should hold between 3 and 7 time units from now and until then $\varphi$ should hold. MTL also contains the notion of *congruences* that allow one to state that a formula should hold periodically with respect to an absolute time. In TLTL there are the additional constructs $\lhd_a \in I$, indicating that the time since $a$ last occurred lies within the interval $I$, and $\rhd_a \in I$, indicating that the time until $a$ next occurs lies within the interval $I$. Whilst these variants of LTL alter the model of traces to include information about time and extend temporal operators to make use of this, they remain inherently regular.

*More Expressive Variants.* We consider further extensions of LTL that increase the expressiveness of the logic as examples of how more complex properties could be captured. There is a rich literature in extending LTL in various ways and this discussion is not meant to be exhaustive.

The first is CaReT [3] which extends LTL with a (context-free) language of *calls* and *returns*. The language is extended with reserved symbols `call` and `ret` annotated with labels for the modules being entered or exited. The temporal operators are then separated into global and abstract forms where the abstract versions reason over the so-called abstract successors of the current position which skips behaviour belonging to nested calls.

Next, one may consider adding *fixed-point operator* to the language, as is done in Eagle [14, 94]. As examples, the maximum fixed point equation $\upsilon x.a \wedge x$ and minimum fixed-point equation $\mu x.a \vee \circ x$ capture the behaviour of $\Box a$ and $\Diamond a$ respectively. Such equations allow behaviour to be defined recursively, which allows context-free behaviour to be captured. In the Eagle setting, the difference between minimum and maximum fixed points is most important when given a finite trace semantics, as it clearly defines what should happen at this boundary.

In [37] Bollig et al. introduce *frequency Linear-time Temporal* (fLTL) which replaces $\mathcal{U}$ with $\mathcal{U}^c$ where $c$ is a rational number between 0 and 1. The formula $\varphi\ \mathcal{U}^c\ \psi$ means that $\varphi$ holds *with frequency* $c$ until $\psi$ holds, meaning that when $c = 1$ this coincides with the standard interpretation of *Until*. The effect of this addition is that fLTL can capture non context-free properties.

As a more exotic example of an expressive variant of LTL is given by Baader et al. [11] who describe a runtime verification approach for a temporal description logic that combines LTL with the *ALC* description logic. As well as allowing description logic axioms to replace axioms, this approach considers the idea of reasoning with *incomplete* information about the trace.

*Signal Temporal Logic.* Another important temporal logic in the runtime verification domain is the setting where the trace is not a discrete sequence of events but a collection of *signals* where a signal is a function from a set of real time points to a value domain. This a setting typically assumed in *hardware monitoring* and comes with its own rich set of specification languages. The standard such language is *Signal Temporal Logic* [117] which includes *signal predicates* of the form $f(x_1[t], \ldots, x_n[t]) > 0$ where $f$ is some function and $x_i[t]$ is the value of the $i$th signal at time $t$. One can use such predicates to define operators to capture the rising and falling edges of a signal. A further defining feature of this logic is the lack of next operator, due to a dense interpretation of time meaning that there is no notion of next state. A consequence of this is that *Until* is typically interpreted with the left operand holding for all times *after* the current point (up until the right operand holds).

*Hyperproperties.* A growing area of interest in RV is that of *hyperproperties* i.e., properties on sets of traces rather than on single traces. There have been various extensions of standard temporal logics to this setting [55] and some have been considered in the context of runtime verification.

**Regular Expressions** A popular declarative language for describing sets of strings in computer science is the regular expression. These have received attention in the runtime verification community, but less attention than temporal logics. We do not spend time describing regular expressions (which should be familiar), but note that they are sometimes used alongside the notion of *suffix-matching* for violations (e.g., in the work of `tracematches` [2]). Later we point out work that combines regular expressions and temporal logic as they are declarative approaches with different advantages. Whilst regular expressions have been extended with a quantitative notion of time [7] and to handle data words [111], such extensions have not received much interest in runtime verification.

**State Machines** Whilst temporal logic and regular expressions are important *declarative* languages for specification they require *monitor synthesis* techniques to produce an executable monitor, which is usually described as some form of state machine. Conversely, state machines have the advantage of being directly executable. As for regular expressions, we do not cover the standard definition of a state machine here, but note that various runtime verification approaches make use of this formalism e.g. [59]. Such approaches do not necessarily agree on exact semantics, but follow the same approach. Areas where approaches may differ include the semantics of completion (what to do if no transition exists), the introduction of various special states, and whether states have explicit output. They may also extend state machines with clocks [59] or deal with extended finite state machines [12]. Some approaches [123] also deal with UML state charts as a state machine representation.

**Beyond Regular** The previous languages were typically regular in nature (with some exceptions). There are also more expressive languages available. This space has not been as well explored, which perhaps suggests that the need for more expressive specification languages is not there, or that such languages have not been accepted for other reasons such as usability.

*Grammars and String Rewriting.* The obvious non-regular language is that of context-free grammars. The key application for such expressiveness is to capture the notion of calls and returns in programs (which can already be handled in the above CaReT logic). A generalised form of grammar is a string rewriting system, which allows arbitrary rewrite rules on strings. Such systems are Turing-complete. JavaMOP [118] includes both context-free grammars and string-rewriting systems as so-called *plugin* languages.

*Rule Systems.* Another powerful formalism is the rule system. In this setting, conditional rules are used to rewrite a set of facts i.e. by adding and removing facts from the set. By predicating a rule on a particular fact, it is then possible to use rules to effectively turn other rules on and off. This setting was first explored in the RuleR system [16] and has been continued in the recent work on LogFire [97].

*Stream Languages.* An alternative approach is to view the trace as one or more *streams* and to define *stream equations* over these streams to produce new streams, which may themselves be the subject of further stream equations. This approach makes computing values (rather than verdicts) over traces straightforward. An early example in this space is LOLA [81].

*Other Approaches.* The above covers the more standard runtime verification approaches. However, there have been various other languages utilised in the field that have received only a little attention. For example, Calder and Sevegnani [43] have utilised a process algebra to perform runtime verification of wireless networks, and Majma et al. [116] make use of coloured petri-nets in their runtime verification of a pacemaker. Recent work [87] makes use of Hennessy-Milner Logic with recursion ($\mu$HML) to describe monitors and explore the monitoring problem in general.

**Combinations** Some specification approaches consider combinations of various languages previously described. Such work aims to find good compromises between the various advantages and disadvantages of different languages.

*Mixing temporal logic and regular expressions.* A popular combination is to add regular expressions to temporal logic. Such a combination typically increases expressiveness (as LTL is *star-free regular*) and make the language more suitable for expressing certain properties involving sequences of events. Examples of combinations include Sugar/PSL [84], RLTL [109], SALT [31], LDL [62], and MDL [25].

*Many in One.* TraceContract [15] provides an internal Scala DSL that supports temporal logic, rule systems, and state machines. As previously mentioned, the JavaMOP tool [118] includes the notion of *plugin* languages which allows users to describe instrumentation in one common language and then use different specification languages over declared events. Supported plugin languages include finite state machines, extended regular expressions, context free grammars, past and future linear temporal logic, CaReT, and string rewriting systems.

*Translations.* As well as combinations of approaches, there are also a number of cases where translations exist between languages. An early example is the embedding of LTL into the very expressive Eagle logic [13]. Other examples include the translation of domain specific languages into more standard logics for runtime verification (e.g. [42]. A recent example is the translation of first-order temporal logic into quantified event automata [127].

### 2.4 Summary

This section has introduced abstractions and languages for describing system behaviour. One conclusion one can draw from this section is that there are a vast number of different ways to describe system behaviour and there is no conclusive silver bullet. Research into specification languages for runtime verification is ongoing and there are many languages that we have not been able to mention in this short summary.

## 3 From Specifications to Monitors

So far we have spoken about how to *specify* desired or undesired system behaviour i.e. a property of a system. In this section we consider the runtime analysis that checks whether a system satisfies or violates a property. We begin by discussing the typical monitoring setup

### 3.1 The Monitoring Setup

As depicted in Fig. 2, a typical RV monitoring setup consists of three main components, namely the system-under-scrutiny, the monitor and the instrumentation mechanism. The collective unit encompassing these three components is then often referred to as the *monitored system*. The previous section discussed how we abstract a system being monitored and here we briefly describe what we mean by a *monitor* and the role of *instrumentation* (although practical instrumentation techniques are discussed in Section 4).

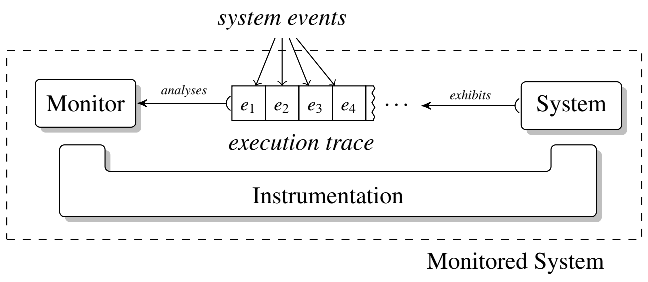


**Fig. 2.** The Basic Monitoring Setup

*Monitors* (execution monitors [131]) are computational entities that execute along side a system so as to *observe* its runtime behaviour and possibly determine whether a property is satisfied or violated from the exhibited (system) execution. When sufficient system behaviour is observed, a monitor may reach a *verdict* (e.g., acceptance or rejection). This verdict is normally assumed to be *definite*, i.e., it cannot be retracted or revised, and is typically communicated to some higher-level entity responsible for handling monitor detections (e.g., the user or some other supervising software component). Whereas few guarantees are expected of the system under scrutiny, in a monitoring setup, monitors are generally considered part of the trusted computing base (TCB) [131, 107, 85] and should manifest a level of correctness themselves. For instance, the verdicts produced by monitors should relate, in some sense, to the property being checked for (e.g., a detected violation should only be flagged by a monitor when the system violates the property being checked) and monitors are also normally expected to interfere minimally (if at all) with the execution of the system under scrutiny. For this reason, monitors are usually generated by *automated synthesis procedures* that take a syntactic representation of the property as input and return the executable code of the monitor as a result. Apart from assisting and expediting monitor construction, automated monitor syntheses mitigate problems associated with the correctness of the monitor itself by standardizing monitor code and giving more scope for a formal treatment of their correctness.

*Instrumentation* is the computational plumbing that connects the execution of a system under scrutiny with the analysis performed by the monitor. It typically concerns itself with two aspects of the monitoring process. First, instrumentation determines what aspects of the system execution are made visible (to the monitor) for analysis. As depicted in Fig. 2, instrumentation records the relevant information from the computation of a running system (e.g., program variable reads/writes, method/function calls and returns, memory management operations such as allocations and deallocations, lock acquisitions and releases, and communication operations such as channel inputs and outputs) and records them as system *events*. Event recording may either consist of redirecting and filtering out existing visible system behaviour, or it may involve extracting aspects of system behaviour that were previously not observable, transforming a black-box system into a grey-box one. The recorded events are then reported to the monitor in the form of an ordered stream called an *execution trace* (of events), which would normally correspond to the same notion introduced in the previous section. Instrumentation usually guarantees that the event order in the execution trace corresponds to the order in which the respective computational step occured. However, there are cases such as in distributed settings where only a *partial ordering* of events can be reliably reported to the monitor.

Second, instrumentation also dictates how the system and the monitor execute in relation to one another in a monitoring setup. For instance, instrumentation may require the system to terminate executing before the monitor starts running, or interleave the respective executions of the system and the monitor that share a common execution thread. In concurrent settings, the system and monitor typically have their own execution threads, and instrumentation may dictates how tightly coupled these executions need to be. The instrumentation may either require that the respective threads to execute synchronously regulated by a global (symbolic) clock [77], or else allow threads to execute asynchronously to one another and then specify synchronisation points between the respective executions. The latter organisation may, in turn, impinge on the timeliness of monitor detections in relation to when the system exhibits a violation to the property being monitored for [46].

Monitoring setups need not necessarily be to confined to the structure and functionality depicted in Fig. 2. In *Monitor-Oriented Programming* (MOP) [49, 50], monitoring is envisaged more as a code design principle advocating for the separation of concerns between the core functionality of a system and ancillary functionality that deals with aspects such as safety, security, reliability and robustness. Code is thus organised as a layered architecture where the innermost core consists of the plain-vanilla system, and the outer layers are made up of monitors observing the execution of the inner layers and reacting to these observations [46]. In MOP, monitors typically do more than just analyse execution traces and raise detections; they may suppress observable behaviour from the inner-layers or filter stimuli coming from outer layers [80, 112, 35], or inject adaptation actions affecting the structure and future behaviour of the inner layers [45, 60, 93].

### 3.2 Monitor Synthesis And Deployment Design Choices

In general, monitoring setups come in various shapes and sizes, and might differ slighly from the clean conceptual view presented in Fig. 2. In what follows, we overview a few of the possible variations commonly encountered in the literature.

**Offline, Online, Synchronous and Asynchronous Monitoring** In offline monitoring (or logging) the analysis is carried out *after* the system executes. Relevant system events are recorded as an execution trace inside a permanent data store (e.g., a file) which is then passed on to the monitor for analysis. Since the execution of the monitor is independent of that of the executing system, an offline analysis is less intrusive and certain constraints such as low monitor runtime overheads do not apply. Offline monitoring also benefits from the fact that the captured execution trace typically describes complete executions, which allows for global trace analyses—these often require backwards traversal [128].

By contrast, online monitoring is performed *during* system execution. It addresses one of the main disadvantages of its offline counterpart, namely that of *late detections*: a violation to a property is only discovered once the system execution terminates, potentially missing the opportunity to mitigate the damage resulting from that violation. Online monitoring operates within tighter constraints such as working with partial executions (i.e., up to the current execution point), stringent requirements for low overheads and, because of this, the need to perform the analysis in incremental fashion.

The simultaneous execution of the system and the monitor may be performed in a variety of ways. At one extreme, *synchronous* online monitoring instruments the system and monitor to execute in lock-step: every time the system generates an event, it waits for the monitor to process it before proceeding with its execution (monitors are by nature passive entities and their execution depends on systems to generate events). At the other extreme, *asynchronous* online monitoring detaches the execution of the monitor from that of the system. This approach is less intrusive on system execution when compared to synchronous monitoring, typically leading to lower overheads [44], but may still yield a degree of late detections (especially when the underlying platform does not guarantee fair executions between the monitor and the system). Due to this, cross-breed approaches that fall on the spectrum in between these two approaches are used to obtain the best of both worlds; consult [47] for a comprehensive survey on this spectrum of approaches.

**Monolithic, Decentralised, Orchestrated and Choreographed Monitor Approaches** There are a number of strategies for synthesising monitors from a particular specification. By far, the most common approach is to synthesise a single monitor that represents the entire specification as one monolithic block (e.g., [30]). Increasingly however, new synthesis strategies are being explored. For instance, the work in [33, 115, 92, 8, 120] synthesise concurrently executing monitors to better exploit the underlying parallel hardware consisting of multiple processing units whereas the work in [78, 32, 9, 67, 68] synthesise component-based monitors to better localise analysis due to multiple event sources and heterogenous hardware. Distribution is another important aspect affecting the monitor synthesis. In general, there are two main strategies for coordinating the monitoring activity across the various distributed locations. Orchestration relies on a single coordinating entity to gather, order and analyse events whereas a choreographed approach disseminates these tasks across a number of monitors [90, 56]. Whereas orchestration is typically simpler to synthesise, thus easier to get right, choreography is more attuned to the characteristics of distributed computing, leading to lower network traffic and a higher degree of fault tolerance [67]. This topic is discussed further in a dedicated chapter.

**Inlining Monitor Code Versus Monitor Code Separation.** Monitoring can be either inlined [70, 135, 51] or consolidated as a separate code unit with events of interest being sent to it (often referred to as outlined). Fig. 2 describes more of a conceptual view rather than the actual implementation, and covers both alternatives. In multi-threaded settings, inlining of inter-thread monitoring requires a *choreographed* setup [135, 90] whereas keeping monitor code separate also affords a centralised *orchestrated* solution. Monitor inlining tends to yield lower overheads and is generally more expressive because it has full access of the system code [70]. By contrast, having monitoring as a separate unit minimally alters the code of the monitored system (all the decision branching is performed inside the monitor), is less error-prone (orchestration tends to be easier to program than monitor choreographies and is harder to tamper with), allows monitor computation to be offloaded to other machines [57], and facilitates compositional analysis whereby monitors may be more readily treated in isolation [85, 89, 1, 86].

## 4 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.

The instrumentation is an important phase in runtime verification setup enabling monitors to be hooked on to the system. 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.). In the following we further explain these concepts in two dedicated sections for hardware and software instrumentation.

### 4.1 Hardware Instrumentation

The increased level of integration, complexity and functionality of the new generation of analog/mixed-signal (AMS) and digital system-on-chip (SoC) technology demands for always new efficient and effective methods to observe and to analyze SoC behaviour both at the physical and at the operational level [4, 5, 102, 103, 122, 130, 132–134].

Due to the complexity of their design, the simulation of such systems is becoming very time-consuming and expensive. For this reason, simulation is generally complemented with design emulation that uses dedicated acceleration platforms such as Field Programmable Gate Arrays (FPGAs) to implement the design under test in hardware. Thus, monitoring the behaviour of an emulated design is an important task supporting the verification of the pre-silicon design. Fig. 3 shows two examples of hardware instrumentation in such scenarios [102]. In the first case (a) the emulated design and the monitor are two independent pieces of hardware. They both share the same source of external clock. The available digital and analog output pins of the emulated design are hooked with physical wires to the hardware monitor. The analog signals are transformed into digital ones using an analog-to-digital (ADC) converter. The obtained signals are then processed synchronously using also a dedicated hardware implementing a monitor. An oscilloscope is employed to observe the verdict of the monitor at runtime. In the second case (b) both the monitor and the design are implemented using the same hardware.

However, simulation and emulation are not able to cover all the aspects of the physical hardware and in particular the software related aspects. In order to check software related problems, large multiprocessor architectures usually require many cycles of executions. In such cases either simulation or emulation may result in too complex and expensive tasks to perform. For this reason, modern SoC have embedded test functionalities, providing a dedicated debug interface [140] called JTAG (also referred as the IEEE Specification 1149.1).

JTAG is a test architecture equipped with a serial interface and other debugging features enabling to sample snapshots of individual SoC pin signals and to drive specific output signals.

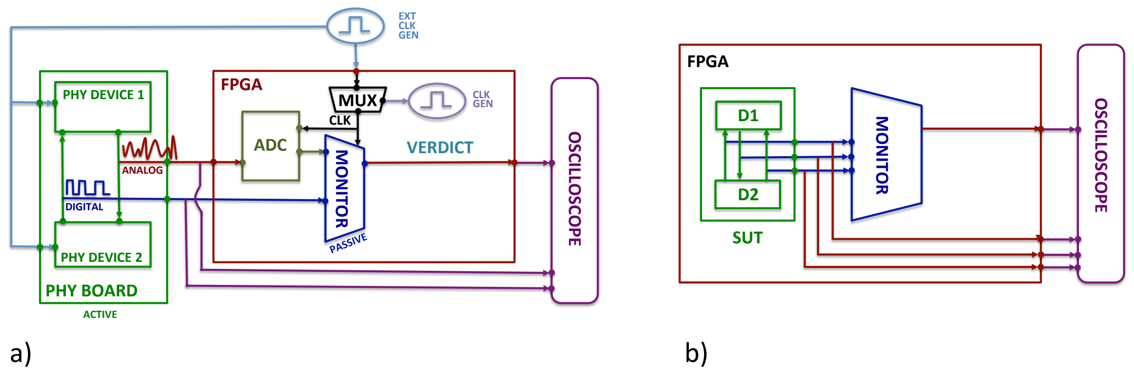


**Fig. 3.** Monitoring design under test: a) the hardware monitor is external to the design under test sharing the same clock generator; b) the emulated design is implemented together with the monitor in the same hardware.

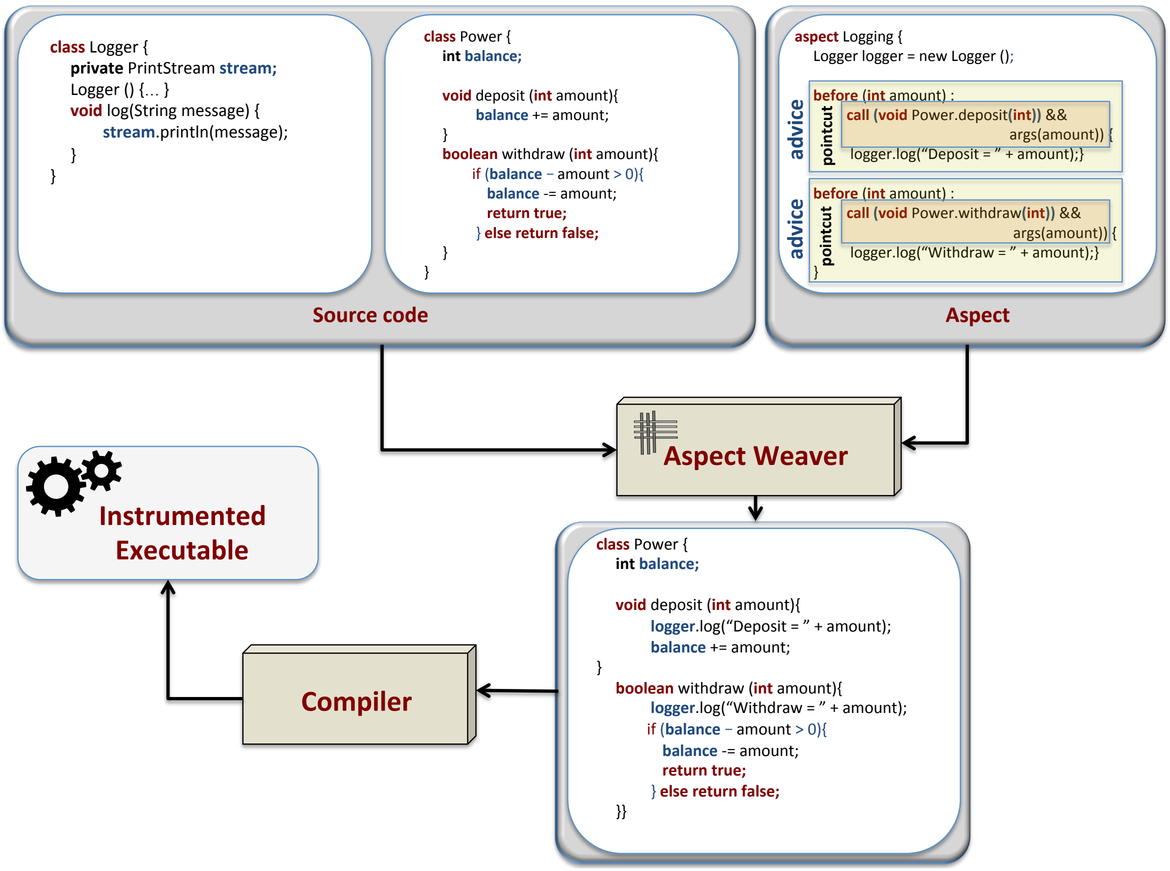
JTAG is nowadays the most popular standard for on-chip instrumentation. Many modern processor architectures such as ARM, x86, MIPS are using JTAG protocol as the foundation for complex data/instruction tracing and debugging. The JTAG port enables the control over the processor that can be halted, single stepped or run freely. However, the possibility to halt the processor running real-time applications can introduce delays in the normal execution altering important timing constraints of the system. For this reason, some designs enable debuggers to access only registers and data buses without the need of halting the processors.

## 4.2 Software Instrumentation

Software instrumentation (SI) is a well-established method employed in many applications including software profiling, performance analysis, optimization, testing and runtime verification. SI consists in adding extra code to track the execution of particular software components and to output an execution trace that can be monitored. The two main approaches for software instrumentation are performed either at the *source code level* [36, 136, 105, 10, 138, 121] or at the *binary level* [34, 40, 114, 119, 108]. Furthermore, SI can be *static* or *dynamic* whether they occur before (i.e., compilation-/link-time) or at execution time (i.e., tracking dynamically linked libraries).

Source code instrumentation consists in adding manually or automatically extra instructions to the software source files before the compilation. Nowadays, there are several instrumentation frameworks [36, 136, 105, 10, 138, 121] available for the main popular programming languages such as Java, C and C++, or even mobile platforms running on Android [72, 73, 66, 61]. For example, *aspect-oriented programming* (AOP) environments usually provide static weaving mechanisms that enable to add at compile-time an additional behaviour to the existing source code without modifying the original source code. The key idea (see Figure 4) is to apply special instructions and code segments (called *advices*) contained in a specification file (*aspect*) that indicate what methods (called *pointcuts*) should be handled by the aspect code. For example, it is possible to specify how to add some additional code to log all the function calls when the function's name starts with a particular prefix. An *aspect weaver* is then the component responsible to process the *advice* instructions and *weave* them together with original source files, generating the final source code that is compiled into an executable. Although in many AOP frameworks the weaving is generally performed statically at the level of source code, there are also cases such as in AspectWerkz [39] where the weaving can occur also dynamically at level of bytecode.

SI is generally limited by the execution coverage. This means that if some parts of the instrumented code are not reachable during the execution, the instrumentation will not provide any information. Furthermore, SI generally introduces a computational overhead that changes the timing-related behaviour of the instrumented program. This could be unacceptable in applications where preserving real-time constraints is extremely important to meet safety critical requirements. In the worst case scenario the overhead of SI may be also the



```
class Logger {
    private PrintStream stream;
    Logger () {... }
    void log(String message) {
        stream.println(message);
    }
}
```

```
class Power {
  int balance;

  void deposit (int amount){
      balance += amount;
  }
  boolean withdraw (int amount){
      if (balance – amount > 0){
        balance -= amount;
        return true;
      } else return false;
  }
}
```

```
aspect Logging {
  Logger logger = new Logger ();

  before (int amount) :
    call (void Power.deposit(int)) &&
                        args(amount)) {
    logger.log("Deposit = " + amount);}

  before (int amount) :
    call (void Power.withdraw(int)) &&
                        args(amount)) {
    logger.log("Withdraw = " + amount);}
}
```

```
class Power {
  int balance;

  void deposit (int amount){
        logger.log("Deposit = " + amount);
        balance += amount;
}
  boolean withdraw (int amount){
        logger.log("Withdraw = " + amount);
      if (balance – amount > 0){
        balance -= amount;
        return true;
      } else return false;
  }}
```

**Fig. 4.** Example of instrumentation of the Java source code with AspectJ.

responsible of timing related *Heisenbugs* [143, 83], bugs that disappear or are altered in the instrumented program. For all these reasons, in the last decade, there has been a great effort to develop new approaches [83, 101, 38, 96, 82, 6] for controlling and mitigating the computational overhead due to SI.

These approaches generally employ sampling-based techniques that reduce the overhead by selecting only a limited and controlled number of events. However, sampling-based techniques are prone to introduce gaps in the output trace, leading to uncertainty in the monitoring result. To quantify such uncertainty, a possible approach (developed in [22, 104, 139]) is to learn statistical models of the monitored system and to use them to fill in sampling-induced gaps in event sequences, and then computing the probability that the property of interest is satisfied or violated.

## 5 Monitorability: What can we monitor?

In this section, we discuss the (notion of) monitorability of properties. Informally, studying the monitorability of a property consists in determining whether or not a property is monitorable, that is, determining whether it is worth monitoring that property at runtime. Intuitively, it is worth monitoring a property if, during monitoring, the monitor can still provide an evaluation (in the form of a verdict) of the current execution and one can avoid situations in which a monitor would inevitably provide inconclusive verdicts.

When using a formalism to write properties (specifying the expected system behavior), one of the questions that arise is whether all properties that are expressible in this formalism can be monitored. Thus, the monitorability question naturally depends on the specification formalism. Moreover, monitorability also depends on the nature of the decision procedure implemented by a runtime monitor. At runtime, a monitor tries to assign a verdict to the current *observation* $\sigma$ (which is by nature finite) by determining whether $\sigma$ is a model or not of the monitored property. Hence, when using a formalism with a semantics over infinite sequences (that is, the models of the properties are infinite executions), monitorability issues arise when relating finite sequences to infinite ones. To intuitively illustrate this issue, consider the two properties "predicate $p$ always holds" and "whenever predicate $p$ holds, predicate $q$ holds in the future". For the first property, suppose that a monitor for this property has in so far observed an execution wherein all observed states predicate $p$ held. In this situation, a monitor is useful because one can later obtain a state leading to the falsification of $p$ and a monitor could detect that an observation wherein $p$ does not hold cannot be extended to the correct infinite correct executions (which are models of the property) wherein predicate $p$ holds at every position. For the second property, suppose that a monitor for this property has in so far observed an execution wherein whenever predicate $p$ held, predicate $q$ held on the next position. Then, suppose that this monitor observes predicate $p$ and receives a certain number of states where predicate $q$ does not hold. Such a monitor can not determine that this sequence cannot be continued to a correct execution. What is worse, the monitor can not relate this execution nor any of its continuations to the infinite ones that are models of the property. Hence, we can see intuitively that it is not worth monitoring that property and the monitor would be doomed producing inconclusive verdicts.

Based on the above informal description of monitorability, several definitions and visions of monitorable properties were proposed. In the rest of this section, we first present the definitions and the associated characterisations of monitorable properties. We note recent research efforts extending the notion of monitorability to decentralised and distributed systems [67]. We focus in the rest of this section to monitorability in the centralised setting, that is following the setup presented in the earlier sections.

## 5.1 Definitions of Monitorability

*Monitoring to detect bad behaviors.* The first definition of monitorability was given by Kim et al. [106]. In their definition, monitoring is purposed to detect violations of properties. They consider safety properties over infinite executions. Since for any safety property, any bad (infinite) execution has a finite prefix that cannot be extended to a correct execution, it is possible to detect any violation of a safety property with a finite sequence. However, the detection of such bad prefixes should be computable. Hence, a property $\varphi \subseteq \Sigma^\omega$ is said to be monitorable if $\Sigma^* \setminus \text{pref}(\varphi)$ is co-recursively enumerable, where $\text{pref}(\varphi)$ is the set of prefixes of $\varphi$.

*Monitoring to detect good and bad behaviors.* Pnueli and Zaks later generalised the notion of monitorability [125]. The underlying principles behind their definition are twofold: a monitor can be also used to detect good behaviors, and a monitor should be running only if it has the possibility to reach a verdict. In their definition, a monitor is purposed to determine whether the current execution leads to infinite continuations that are models of the monitored property. A monitor can then determine a verdict whenever either every continuation of the current observation is a model or every continuation is a counter-example of the monitored property. Moreover, it is worth monitoring as long as the monitor can find a verdict with a possible continuation of the observed sequence.

More precisely, their definition of monitorability comes as follows. For a property $\varphi \subseteq \Sigma^\omega$, and given the current execution $\sigma \in \Sigma^*$, $\varphi$ is said to be positively determined by $\sigma$ if all (infinite) continuations of $\sigma$ satisfy $\varphi$. Conversely, Pnueli and Zaks also define the notion of negative determinacy. Whenever an execution $\sigma$ positively (resp. negatively) determines a property $\varphi$, a monitor for $\varphi$ associates verdict $\top$ (resp. $\bot$) with $\sigma$. Then, $\varphi$ is said to be $\sigma$-monitorable, if $\sigma$ has a continuation such that $\varphi$ is either positively or negatively determined by this continuation. Finally, a property is monitorable if it is $\sigma$-monitorable, for any $\sigma \in \Sigma^*$.

We note that in [64], Diekert and Leucker provide an equivalent topological definition of monitorability. Given a property to monitor $\varphi$ over some alphabet $\Sigma$ and an execution sequence $\sigma \in \Sigma^*$, $\varphi$ is monitorable at $\sigma$ (alternatively $\sigma$-monitorable) if every open set $O_\sigma$ containing $\sigma$ has a subset $SO_\sigma$ such that either $SO_\sigma \subseteq \varphi$ or $SO_\sigma \subseteq (\Sigma^* \setminus \varphi)$, which one can understand as $\sigma$ has a continuation that positively or negatively determines $\varphi$. And then $\varphi$ is monitorable if it is monitorable at every $\sigma \in \Sigma^*$.

*Monitoring with a parameterised verdict domain.* The previous definition of monitorability implicitly uses the 3-valued truth-domain $\{\bot, ?, \top\}$ where $\bot$ and $\top$ are final verdicts (i.e., assigned once and never changed) and verdict ? is emitted by the monitor for other sequences (i.e., those not allowing it to reach a final verdict). Falcone et al. argue that in some situations, one may monitor only to detect satisfactions or violations in separation [75]. Hence, they parameterised the definition of monitorability by a truth-domain $\mathbb{B}$ that contains at least one final verdict.

*Monitoring with a semantics for finite executions.* To account for the situations where the monitored program stops before the monitor is able to reach a final verdict (i.e., when the last verdict is ?), Falcone et al. introduce a notion of monitorability taking finite executions into account [75]. The definitions requires the specification formalism to be endowed with a semantics over finite sequences including verdicts $\top_c$ and $\bot_c$ used to indicate that the property is *currently true* and *currently false*, respectively. An execution sequence evaluates to $\top$ and $\bot$ as in the definition of Pnueli et al. [125], and it evaluates to $\top_c$ (resp. $\bot_c$) as long as no definitive verdict has been found and the current execution sequence satisfies (resp. does not satisfy) the property. Intuitively, for a property to be monitorable, the evaluations of the property on correct and incorrect finite executions should differ so that a monitor is able to detect in a sound manner the situations in which it should emit a final verdict.

*Monitoring for a branching-time logic.* Francalanza et al. [88, 89] study the problem of monitoring for branching-time logics and define $\mu$-HML a reformulation of $\mu$-calculus as a branching-time logic with least and greatest fix-points. Monitorability of a formula in this logic amounts to being able to synthesise a sound monitor that is able to detect all violations or all satisfactions of the formula. A salient aspect of this body of work is the identification of a maximally-expressive syntactic subset of the logic whereby any monitorable property is guaranteed to be expressible within this syntactic subset (similar maximality guarantees are also given in the context of LTL in [54]). More importantly, the branching nature of the logic considered gives scope for considering monitoring setups that depart from the classic setup consisting of one system execution generating a single trace, since alternative setups may extend the set of monitorable properties.

## 5.2 Characterisations of Monitorable Properties

We now report on the existing characterisations of monitorable properties. Characterising monitorable properties as a class of properties is important because, when specifying a system, it allows determining the monitorability of the specified property just by determining the class to which the property belongs (for instance using the syntactic elements used to construct the property).

*Characterisation for the definition in [106].* Kim et al. directly define monitorable properties as the class of safety properties such that the set of prefixes is co-recursively enumerable.

*Characterisation for the definition in [125].* The definition of monitorability in [125] is the most studied one. Bauer et al. [28, 30] prove that the set of monitorable properties in the sense of [125] is a (strict) superset of the union of safety and co-safety properties. Falcone et al. [74, 75] prove that the set of monitorable properties in the sense of [125] is a (strict) super set of the set of obligation properties (which are formed by finite conjunctions and disjunctions of safety and co-safety properties). They also prove that adding additional verdicts to the definition of monitorability in [125] does not allow monitoring more properties. Later in [64], Diekert and Leucker enunciate the same results as in [75] from a topological perspective. They additionally prove that any countable union/disjunction or any countable intersection/conjunction of monitorable sets/properties is also monitorable. In [65] Diekert et al. study the complexity of deciding monitorability. They show that i) deciding whether a Büchi automaton defines a monitorable property is PSPACE-complete, and ii) deciding whether an LTL formula defines a monitorable property is PSPACE-hard and EXPSPACE-easy.

*Characterisation for the definition in [75].* Falcone et al. [75] prove that the set of monitorable properties in the sense of [75] forms a strict subset of obligation properties. They also prove that the definition in [75] allows monitoring any (linear-time) property when used with truth-domain $\{\bot, \bot_c, \top_c, \top\}$.

## 6 Overview of the Handbook

The idea of this book originated from the need to have an handbook for students to support their training with several tutorials on different aspects of RV. The volume has been organized into seven chapters. This chapter can be considered a primer to the field and necessary knowledge for the rest of this book.

The second chapter [113] is dedicated to the detection of concurrency errors raised in concurrent programming. The chapter presents how dynamic analysis techniques can be used for the detection and localisation of data races, atomicity violations, and deadlocks.

The third chapter [100] shows to adapt early-stage runtime verification frameworks wherein events are names to events that carry date. The chapter shows how adding data to events complexifies the specification language and the underlying monitoring algorithms. The chapter overviews five specification formalisms and associated monitoring algorithms.

The fourth chapter [79] presents how runtime monitors can be used to prevent and react to failures to increase the dependability of systems. For this, it presents the two main techniques for such purposes, namely runtime enforcement and healing failures, respectively.

The fifth chapter [20] revolves around the techniques for the monitoring of specifications on cyber-physical systems. The behaviour of cyber-physical systems is modeled by continuous state variables interleaved with discrete events. The chapter presents state-of-the-art techniques for using qualitative and quantitative monitoring techniques either during simulation or when the system is running. The chapter also presents example applications and compares exiting tools.

The sixth chapter [91] tackles the emerging and important topics of decentralised monitoring and distributed monitoring. The chapter identifies the distinguishing features of decentralised and distributed systems and classifies the existing approaches along these features.

The seventh chapter [58] is dedicated to the application of runtime verification to industrial systems and more particularly on financial transaction systems. This chapter places runtime verification in the development lifecycle of a software. It interestingly describes some of the properties that can be useful in real-life applications. Moreover, it reports on some of the lessons learned by the authors and outlines some of the challenges to address for RV to become an industrial practice.

## 7 Conclusion

This chapter has given a brief introduction to the field of runtime verification covering four major topics: how to specify system behaviour, how to setup monitoring, how to perform instrumentation, and what the limitations of monitoring are. We refer the reader to the other chapters in this book and other introductions to RV [71, 110] for further details on the topic.

6. Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie du Bousquet. Compressing microcontroller execution traces to assist system analysis. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro Cesar Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 139–150. Springer, 2013.
7. Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, and Lydie du Bousquet. Fault localization in embedded software based on a single cyclic trace. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 148–157. IEEE Computer Society, 2013.
8. Matthew Arnold, Martin T. Vechev, and Eran Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *Proc. of OOPSLA 2008: the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 143–162. ACM, 2008.
9. Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49:2002, 2001.
10. Duncan Paul Attard and Adrian Francalanza. A Monitoring Tool for a Branching-Time Logic. In *Proc. of RV'16*, volume 10012 of *LNCS*, pages 473–481. Springer, 2016.
11. Duncan Paul Attard and Adrian Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, volume 10469 of *LNCS*, pages 219–235, 2017.
12. Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc* : An extensible aspectj compiler. *Transactions on Aspect-Oriented Software Development I*, 3880:293–334, 2006.
13. Franz Baader, Andreas Bauer, and Marcel Lippmann. Runtime verification using a temporal description logic. In *Proc. of FroCoS 2009: the 7th International Symposium on Frontiers of Combining Systems*, volume 5749 of *LNCS*, pages 149–164. Springer, 2009.
14. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.
15. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004.
16. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
17. Howard Barringer and Klaus Havelund. Tracecontract: A scala DSL for trace analysis. In *Proc. of FM 2011: the 17th International Symposium on Formal Methods, Limerick*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
18. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.
19. Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In *Proc. of RV 2014: the 5th International Conference on Runtime Verification*, volume 8734 of *LNCS*, pages 1–9. Springer, 2014.
20. Ezio Bartocci, Flavio Corradini, Emanuela Merelli, and Luca Tesei. Model checking biological oscillators. *Electr. Notes Theor. Comput. Sci.*, 229(1):41–58, 2009.
21. Ezio Bartocci, Flavio Corradini, Emanuela Merelli, and Luca Tesei. Detecting synchronisation of biological oscillators by model checking. *Theor. Comput. Sci.*, 411(20):1999–2018, 2010.
22. Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
23. Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 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.
24. Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. Adaptive runtime verification. In *Proc. of RV 2012: the Third International Conference on Runtime Verification*, volume 7687 of *LNCS*, pages 168–182. Springer, 2013.
25. Ezio Bartocci and Rupak Majumdar, editors. *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*. Springer, 2015.
26. Ezio Bartocci and Rupak Majumdar, editors. *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*. Springer, 2015.
27. David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

28. David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Runtime verification of temporal properties over out-of-order data streams. In Rupak Majumdar and Viktor Kuncak, editors, *CAV 2017*, volume 10426 of *LNCS*, pages 356–376. Springer, 2017.
29. David A. Basin, Srdan Krstic, and Dmitriy Traytel. Almost event-rate independent monitoring of metric dynamic logic. In *Proc. of RV 2017: the 17th International Conference on Runtime Verification*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
30. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. The ins and outs of first-order runtime verification. *Formal Methods in System Design*, 46(3):286–316, 2015.
31. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proc. of RV 2007: the 7th International Workshop on Runtime Verification*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.
32. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007.
33. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.
34. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.
35. Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT - structured assertion language for temporal logic. In *Proc. of ICFEM 2006: the 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.
36. Andreas Klaus Bauer and Yliès Falcone. Decentralised ltl monitoring. *Formal Methods in System Design*, 48(1–2):49–93, 2016.
37. Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. Runtime verification with minimal intrusion through parallelism. *Formal Methods in System Design*, 46(3):317–348, 2015.
38. Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proc. of PASTE'11: the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16. ACM, 2011.
39. Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced?: Edited automata revisited. *Int. J. Inf. Secur.*, 10(4):239–254, August 2011.
40. Eric Bodden and Klaus Havelund. Racer: Effective race detection using aspectj. In *Proc. of ISSTA '08: the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 155–166. ACM, 2008.
41. Benedikt Bollig, Normann Decker, and Martin Leucker. Frequency Linear-time Temporal Logic. In *Proceedings of the 6th International Symposium on Theoretical Aspects of Software Engineering (TASE'12)*, pages 85–92, Beijing, China, 2012. IEEE Computer Society Press.
42. Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *Proc. of FM 2011: the 17th International Symposium on Formal Methods*, volume 6664 of *LNCS*, pages 88–102. Springer, 2011.
43. Jonas Bonér. What are the key issues for commercial AOP use: how does aspectwerkz address them? In *Proc. of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*, pages 5–6. ACM, 2004.
44. Manfred Broy, Doron A. Peled, and Georg Kalus, editors. *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*. IOS Press, 2013.
45. Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of CGO 2003: the 1st IEEE / ACM International Symposium on Code Generation and Optimization*, pages 265–275. IEEE Computer Society, 2003.
46. Tevfik Bultan and Koushik Sen, editors. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. ACM, 2017.
47. Aaron Calafato, Christian Colombo, and Gordon J. Pace. A controlled natural language for tax fraud detection. In *Proc. of CNL 2016: the 5th International Workshop on Controlled Natural Language*, volume 9767 of *LNCS*, pages 1–12. Springer, 2016.
48. Muffy Calder and Michele Sevegnani. Process algebra for event-driven runtime verification: A case study of wireless network management. In *Proceedings of the 9th International Conference on Integrated Formal Methods*, IFM'12, pages 21–23, Berlin, Heidelberg, 2012. Springer-Verlag.
49. Ian Cassar and Adrian Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In Proceedings 13th International Workshop on *Foundations of Coordination Languages and Self-Adaptive Systems,* Rome, Italy, 6th September 2014, volume 175 of *Electronic Proceedings in Theoretical Computer Science*, pages 54–68. Open Publishing Association, 2015.

50. Ian Cassar and Adrian Francalanza. Runtime Adaptation for Actor Systems. In *Runtime Verification (RV)*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.
51. Ian Cassar and Adrian Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192. Springer, 2016.
52. Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *PrePost@iFM*, volume 254 of *EPTCS*, pages 15–28, 2017.
53. Antonio Cau and Hussein Zedan. Refining interval temporal logic specifications. In *Proc. of ARTS'97: th 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, volume 1231 of *LNCS*, pages 79–94. Springer, 1997.
54. Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *ENTCS*, pages 106–125. Elsevier, 2003.
55. Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.
56. Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588. ACM press, 2007.
57. Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *TACAS 2009*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.
58. Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.
59. Clare Cini and Adrian Francalanza. An LTL Proof System for Runtime Verification. In *TACAS*, volume 9035, pages 581–595. Springer, 2015.
60. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of POST 2014: the Third International Conference on Principles of Security and Trust*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
61. Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.
62. Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace. polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In *SEFM*, pages 218–232, 2012.
63. Christian Colombo and Gordon J. Pace. Industrial experiences with runtime verification of financial transaction systems: Lessons learnt and standing challenges. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
64. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *Proc. of SEFM 2009: the Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37. IEEE Computer Society, 2009.
65. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-adaptive monitors for multiparty sessions. In *PDP*, pages 688–696. IEEE Computer Society, 2014.
66. Philip Daian, Yliès Falcone, Patrick O'Neil Meredith, Traian-Florin Serbanuta, Shin'ichi Shiriashi, Akihito Iwai, and Grigore Rosu. Rv-android: Efficient parametric android runtime verification, a brief tutorial. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2015.
67. Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 854–860. AAAI Press, 2013.
68. Normann Decker and Daniel Thoma. On freeze LTL with ordered attributes. In Bart Jacobs and Christof Löding, editors, *FoSSaCS 2016*, volume 9634 of *LNCS*, pages 269–284. Springer, 2016.
69. Volker Diekert and Martin Leucker. Topology, monitorable properties and runtime verification. *Theor. Comput. Sci.*, 537:29–41, 2014.
70. Volker Diekert, Anca Muscholl, and Igor Walukiewicz. A note on monitors and Büchi automata. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2015.
71. Khalil El-Harake, Yliès Falcone, Wassim Jerad, Matthieu Langet, and Mariem Mamlouk. Blocking advertisements on android devices using monitoring techniques. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2014.

72. Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In Bultan and Sen [46], pages 125–135.
73. Antoine El-Hokayem and Yliès Falcone. THEMIS: a tool for decentralized monitoring algorithms. In Bultan and Sen [46], pages 372–375.
74. E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. MIT Press Cambridge, 1990.
75. Ulfar Erlingsson. *The Inlined Reference Monitor approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.
76. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In Manfred Broy and Doron Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
77. Yliès Falcone and Sebastian Currea. Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 350–353. ACM, 2012.
78. Yliès Falcone, Sebastian Currea, and Mohamad Jaber. Runtime verification and enforcement for android applications with rv-droid. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*, pages 88–95. Springer, 2012.
79. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *Proc. of RV 2009: the 9th International Workshop on Runtime Verification*, volume 5779 of *LNCS*, pages 40–59. Springer, 2009.
80. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.
81. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
82. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems. In *SEFM*, volume 7041 of *LNCS*, pages 204–220. Springer, 2011.
83. Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the bip framework with formally-proved sound and complete instrumentation. *Software & Systems Modeling*, 14(1):173–199, 2015.
84. Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime error reaction and prevention. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
85. Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.
86. Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In Yliès Falcone and César Sánchez, editors, *RV 2016*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.
87. Long Fei and Samuel P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. In *Proc. of the ACM SIGPLAN 2006: Conference on Programming Language Design and Implementation*, pages 84–95. ACM, 2006.
88. Sebastian Fischmeister and Patrick Lam. On time-aware instrumentation of programs. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, RTAS '09, pages 305–314, Washington, DC, USA, 2009. IEEE Computer Society.
89. H. Q. Foster, Erich Marschner, and Yaron Wolfsthal. Ieee 1850 psl: The next generation. 2005.
90. Adrian Francalanza. A Theory of Monitors. In *FoSSaCS*, volume 9634 of *LNCS*, pages 145–161. Springer, 2016.
91. Adrian Francalanza. Consistently-Detecting Monitors. In *CONCUR*, volume 85 of *LIPIcs*, pages 8:1–8:19, 2017.
92. Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *Proc. of RV 2017: the 17th International Conference on Runtime Verification*, volume 10548 of *LNCS*, pages 8–29. Springer, 2017.
93. Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On verifying hennessy-milner logic with recursion at runtime. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2015.
94. Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy-Milner Logic with Recursion. *Journal of Formal Methods in System Design (FMSD)*, 2017. (to appear).
95. Adrian Francalanza, Andrew Gauci, and Gordon J. Pace. Distributed System Contract Monitoring. *JLAP*, 82(5-7):186–215, 2013.

96. Adrian Francalanza, Jorge A. Perez, and Cesar Sanchez. Runtime verification for decentralized and distributed systems. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
97. Adrian Francalanza and Aldrin Seychell. Synthesising Correct concurrent Runtime Monitors. *FMSD*, 46(3):226–261, 2015.
98. Cinzia Di Giusto and Jorge A. Perez. Disciplined Structured Communications with Disciplined Runtime Adaptation. *Sci. of Computer Programming*, 97(2):235–265, 2015.
99. Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors. *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. ACM, 2012.
100. Allen Goldberg and Klaus Havelund. Automated runtime verification with eagle. In *Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2005, In conjunction with ICEIS 2005, Miami, FL, USA, May 2005*, 2005.
101. Sylvain Hallé and Roger Villemaire. Runtime monitoring of message-based workflows with data. In *ECOC 2008*, pages 63–72. IEEE Computer Society, 2008.
102. Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of ASPLOS: the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164. ACM, 2004.
103. Klaus Havelund. Rule-based runtime verification revisited. *STTT*, 17(2):143–170, 2015.
104. Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *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*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.
105. Klaus Havelund and Giles Reger. Runtime verification logics - a language design perspective. In *KIMfest - a conference in honour of Kim G. Larsen on the occasion of his 60th birthday, Aalborg University, 19-20 August 2017*, volume TBD of *LNCS*. Springer, 2017.
106. Klaus Havelund, Giles Reger, Eugen Zalinescu, and Daniel Thoma. Monitoring events that carry data. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
107. Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *STTT*, 14(3):327–347, 2012.
108. Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Ničković. From signal temporal logic to FPGA monitors. In *Proc. of MEMOCODE 2015: the 13th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 218–227. IEEE, 2015.
109. Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Nickovic. Quantitative monitoring of STL with edit distance. In *Proc. of RV 2016: the 16th International Conference on Runtime Verification*, volume 10012 of *LNCS*, pages 201–218, 2016.
110. Kenan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott D. Stoller, and Radu Grosu. Runtime verification with particle filtering. In *Proc. of RV 2013: the 4th International Conference on Runtime Verification*, volume 8174 of *LNCS*, pages 149–166. Springer, 2013.
111. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proc. of ECOOP 2001: the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
112. Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electr. Notes Theor. Comput. Sci.*, 70(4):80–94, 2002.
113. Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the Guardians. In *RV*, volume 9333 of *LNCS*, pages 87–101. Springer, 2015.
114. Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 175–183. IEEE Computer Society, 2010.
115. Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors. *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*. Springer, 2015.
116. Martin Leucker and César Sánchez. Regular linear temporal logic. In *Proc. of ICTAC 2007: the 4th International Colloquium on Theoretical Aspects of Computing*, volume 4711 of *LNCS*, pages 291–305. Springer, 2007.
117. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
118. Leonid Libkin and Domagoj Vrgoc. Regular expressions for data words. In *Proc. of LPAR-18: the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2012.

119. Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 4(1-2):2–16, 2005.
120. João Lourenço, Jan Fiedor, Bohuslav Krena, and Tomás Vojnar. Discovering concurrency errors. In *Handbook of Runtime Verification - Introductory and Advanced Topics*, volume 10475 of *LNCS*. 2018.
121. Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN 2005: the Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005.
122. Qingzhou Luo and Grigore Roşu. EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs. In *ISSTA*, New York, NY, USA, 2013. ACM.
123. N. Majma, S. M. Babamir, and A. Monadjemi. Runtime verification of pacemaker using fuzzy logic and colored petri-nets. In *2015 4th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)*, pages 1–5, Sept 2015.
124. Oded Maler and Dejan Ničković. Monitoring temporal properties of continuous signals. In *Proc. of FORMATS/FTRTFT 2004: Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.
125. Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*. Springer, 2014.
126. Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
127. Bertrand Meyer and Jim Woodcock, editors. *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*, volume 4171 of *Lecture Notes in Computer Science*. Springer, 2008.
128. Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors. *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*. Springer, 2006.
129. Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: binary interpretation using runtime disassembly. In *Proc. of CGO 2006: the Fourth IEEE/ACM International Symposium on Code Generation and Optimization*, pages 358–370. IEEE Computer Society, 2006.
130. Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, and Marius Bozga. Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. *Formal Asp. Comput.*, 29(6):951–986, 2017.
131. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.
132. Thang Nguyen, Ezio Bartocci, Dejan Ničković, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In *Proc. of ISoLA 2016: 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 9953 of *LNCS*, pages 371–379. Springer, 2016.
133. Gergely Pintér and István Majzik. Runtime verification of statechart implementations. In *Architecting Dependable Systems III [the book is a result of the ICSE 2004 and DSN 2004 workshops]*, volume 3549 of *LNCS*, pages 148–172. Springer, 2005.
134. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
135. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.
136. Shaz Qadeer and Serdar Tasiran, editors. *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*. Springer, 2013.
137. Giles Reger and Klaus Havelund. What is a trace? a runtime verification perspective. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Disc ussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, pages 339–355. Springer International Publishing, 2016.

138. Giles Reger and David E. Rydeheard. From first-order temporal logic to parametric trace slicing. In Ezio Bartocci and Rupak Majumdar, editors, *RV 2015*, volume 9333 of *LNCS*, pages 216–232. Springer, 2015.
139. Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.
140. Grigore Roşu and Saddek Bensalem. Allen linear (interval) temporal logic –translation to ltl and monitor synthesis–. In *Proceedings of 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.
141. Selma Saidi and Yliès Falcone. Dynamic detection and mitigation of DMA races in mpsocs. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 267–270. IEEE Computer Society, 2015.
142. Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro Cesar Zanella, and Franz J. Rammig, editors. *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, volume 403 of *IFIP Advances in Information and Communication Technology*. Springer, 2013.
143. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.
144. Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime monitoring with recovery of the SENT communication protocol. In *Proc. of CAV 2017: the 29th International Conference on Computer Aided Verification*, volume 10426 of *LNCS*, pages 336–355. Springer, 2017.
145. Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. Applying runtime monitoring for automotive electronic development. In *Proc. of RV 2016: the 16th International Conference on Runtime Verification*, volume 10012 of *LNCS*, pages 462–469. Springer, 2016.
146. Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, Dejan Ničković, and Radu Grosu. Monitoring of MTL specifications with ibm's spiking-neuron model. In *Proc. of DATE 2016: the 2016 Design, Automation & Test in Europe Conference*, pages 924–929. IEEE, 2016.
147. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. *ICSE*, pages 418–427, 2004.
148. Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Interaspect: aspect-oriented instrumentation with GCC. *Formal Methods in System Design*, 41(3):295–320, 2012.
149. Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.
150. Oleg Sokolsky and Serdar Tasiran, editors. *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*. Springer, 2007.
151. Olaf Spinczyk and Daniel Lohmann. The design and implementation of aspectc++. *Know.-Based Syst.*, 20(7):636–651, October 2007.
152. Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *Proc. of RV 2011: the 2nd International Conference on Runtime Verification*, volume 7186 of *LNCS*, pages 193–207. Springer, 2012.
153. Neal Stollon. *On-Chip Instrumentation*. Springer US, 2011.
154. Volker Stolz. Temporal assertions with parametrized propositions. *J. Log. Comput.*, 20(3):743–757, 2010.
155. Prasanna Thati and Grigore Ro u. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145 – 162, 2005.
156. Marianne Winslett. Bruce lindsay speaks out: On system r, benchmarking, life as an ibm fellow, the power of dbas in the old days, why performance still matters, heisenbugs, why he still writes code, singing pigs, and more. *SIGMOD Rec.*, 34(2):71–79, June 2005.
157. Shikun Zhou, Hussein Zedan, and Antonio Cau. Run-time analysis of time-critical systems. *J. Syst. Archit.*, 51(5):331–345, May 2005.